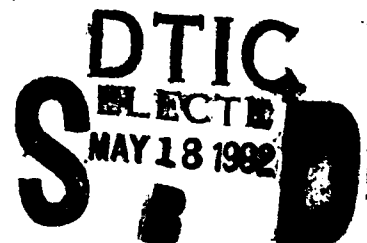# NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A250 703

THESIS

Adding Intelligence to the Composite Warfare
Commander - Distributed Dynamic
Decisionmaking (CWC-DDD) Paradigm

by

Brian Kenneth Wright

March 1992

Thesis Advisor: Kishore Sengupta

92-12957

92 5 14 078

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION		1b. RESTRICTIVE MARKINGS
UNCLASSIFIED		

2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
	Approved for public release; distribution is unlimited.
2b. DCLASSIFICATION/DOWNGRADING SCHEDULE	

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If Applicable)	7a. NAME OF MONITORING ORGANIZATION
Administrative Sciences Department Naval Postgraduate School	AS	Naval Postgraduate School

6c. ADDRESS (city, state, and ZIP code)	7b. ADDRESS (city, state, and ZIP code)
Monterey, CA 93943-5000	Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	6b. OFFICE SYMBOL (If Applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (city, state, and ZIP code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
ADDING INTELLIGENCE TO THE COMPOSITE WARFARE COMMANDER - DISTRIBUTED DYNAMIC DECISIONMAKING (CWC-DDD) PARADIGM

12. PERSONAL AUTHOR(S)
Brian Kenneth Wright

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Master's Thesis	FROM 08/90 TO 03/92	March 1992	104

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17.	COSATI CODES		18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUBGROUP	Decisionmaking, Distributed Decisonmaking, CWC, C Programming,
			Sun Workstations, Sunview, War Games

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm is a tool for experimentation and research into the area of command, control and communications (C3) team decisionmaking process in simulated Navy engagement scenarios. It is implemented as a computer-driven interactive game among four person hierarchical teams of decisionmakers on a network of worstations. The paradigm is a compromise between controllability and realism of the experimental environment. The major drawback with the current implementation is the lack of responsiveness of the tasks (attackers) to the actions of the assets (defenders) and the environmental conditions. This thesis details ways to improve the responsiveness of the attackers and the realism of the paradigm by the implementation of a group of if-then heuristics. The five proposed heuristics are designed to make the attackers attempt to evade the defenders while still actively pursuing their mission to penetrate the center of the battle group. The heuristics are implemented in the RAINCOAT version of the paradigm using the C programming language. The heuristics are validated by several military commanders for adherence with the accepted battle doctrine of the Navy's Composite Warfare Command.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Dr. Kishore Sengupta	(408) 646- 3212	AS/Se

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete   Unclassified

Adding Intelligence to the Composite Warfare Commander - Distributed
Dynamic Decisionmaking Paradigm

by

Brian Kenneth Wright
Lieutenant, United States Coast Guard
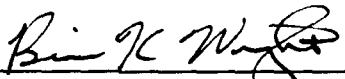**B.S., United States Coast Guard Academy, 1983**

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS
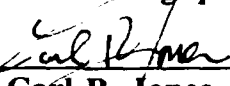
from the

NAVAL POSTGRADUATE SCHOOL
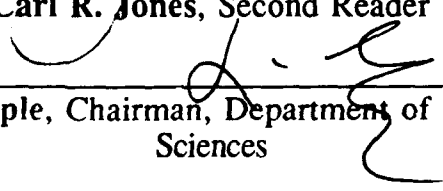**March 1992**

Author: _____
Brian Kenneth Wright

Approved by: _____
Kishore Sengupta, Thesis Advisor

_____
Carl R. Jones, Second Reader

_____
David R. Whipple, Chairman, Department of Administrative
Sciences

ii

ABSTRACT

The Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm is a tool for experimentation and research into the area of command, control and communications (C3) team decisionmaking process in simulated Navy engagement scenarios. It is implemented as a computer-driven interactive game among four person hierarchical teams of decisionmakers on a network of workstations. The paradigm is a compromise between controllability and realism of the experimental environment. The major drawback with the current implementation is the lack of responsiveness of the tasks (attackers) to the actions of the assets (defenders) and the environmental conditions. This thesis details ways to improve the responsiveness of the attackers and the realism of the paradigm by the implementation of a group of if-then heuristics. The five proposed heuristics are designed to make the attackers attempt to evade the defenders while still actively pursuing their mission to penetrate the center of the battle group. The heuristics are implemented in the RAINCOAT version of the paradigm using the C programming language. The heuristics are validated by several military commanders for adherence with the accepted battle doctrine of the Navy's Composite Warfare Command.

iii

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I. INTRODUCTION

A. PURPOSE OF THE RESEARCH

The Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm was developed as a tool for experimentation and research into the area of command, control and communication (C3) team decisionmaking processes in simulated combat situations. It has been used extensively in a series of research projects commissioned by the Office of Naval Research. Since the paradigm will be the basis for additional experiments in this subject area, it is important that the paradigm be examined from a variety of perspectives and improvements made whenever possible. This research project was an examination of the paradigm from the perspective of junior military officers.

B. RESEARCH METHODOLOGY

The research consisted of three primary phases. First, a series of 42 experimental scenarios were run using military officers from the Joint Command, Control and Communication curriculum at the Naval Postgraduate School as subjects. Second, the strengths and weaknesses of the paradigm were discussed with the subjects and several impartial observers through a combination of informal discussions and written questionnaires. Third, a set of heuristics that would "add intelligence" to the enemy in the paradigm were developed and implemented where possible.

1

C. POTENTIAL BENEFITS OF THE RESEARCH

The CWC-DDD paradigm will continue to be used as a tool to research team distributed dynamic decisionmaking processes. An improved paradigm will be closer to the "real world" and, therefore, should result in experimental findings that are more transferable to the real world. Additionally, the review of the paradigm from a military perspective and the addition of intelligence heuristics will serve as valuable feedback to the developers of the paradigm.

D. HISTORY OF THE CWC-DDD PARADIGM

Current Naval doctrine for the defense of a battle group is based upon the doctrine of the Composite Warfare Command. Under this doctrine, several warfare commanders, who may be geographically separated, are charged with defending the battle group against a variety of air, surface, and subsurface threats. They must coordinate their actions and pool their resources to be successful in their defense of the battle group. (Shi, Luh and Kleinman, 1990, p. 48)

Generally, there are four warfare commanders. The Composite Warfare Commander (CWC) has overall responsibility for the defense of the battle group. To aid him in this, he has three subordinate warfare commanders. The first subordinate warfare commander is the Anti-aircraft Warfare Commander (AAWC). He is responsible for defense in the air arena. The second subordinate warfare commander is the Anti-surface Warfare Commander (ASWC). He is responsible for defense in the surface arena. The third subordinate warfare commander is the Anti-submarine Warfare Commander (ASuWC). He is responsible for defense in the subsurface arena.

The Navy has identified the issues of conflict resolution and resource contention within the Composite Warfare Command doctrine as two of the more serious problems facing the Naval Command and Control (C2) structure (Shi, Luh and Kleinman, 1990, p. 48). In an effort to better understand the dynamics of the distributed decisionmaking process, the Office of Naval Research (ONR) contracted with the University of Connecticut (UCONN) and Alphatech, Inc., to complete a study of this process. The objective of this initial study and the follow-on studies was to not only develop a better understanding of this decisionmaking process, but to arrive at methods to better train Naval commanders.

UCONN and Alphatech took a normative-descriptive approach to the problem. First, they abstracted the "real world" problems to bring them into the controlled laboratory environment where a variety of experimental conditions could be manipulated individually. Second, they completed empirical and analytical studies of human team decisionmaking in order to develop a model of the distributed dynamic decisionmaking process. Most of their theoretical work was based upon the five years of basic experimental research done by Dr. David Kleinman on the more general Distributed Dynamic Decisionmaking (DDD) paradigm. Most of the CWC-DDD software programming was done by Dr. Anlan Song. (Kleinman and Song, 1990, p. 129)

The result of their efforts was the Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm. It was designed to support the ongoing efforts to examine distributed decisionmaking issues in four person hierarchical teams of naval commanders. The issues that can be examined include planning, coordinating, and allocating resources in a relatively

realistic naval engagement. The paradigm was designed to have the flexibility to examine the variety of ways which information processing and resource allocation problems can be handled by teams under different environmental constraints. (Kleinman and Song, 1990, p. 129)

II. THE CWC-DDD PARADIGM

A. INTRODUCTION

The Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm is implemented as a computer-driven interactive game among several decisionmakers (DMs) on a network of Sun workstations. It attempts to simulate "real world" Navy engagement scenarios faced by the Composite Warfare Commander and his subordinates (Kleinman and Song, 1990, p. 129).

The game is played a team of up to four members. One DM functions as leader, and the other DMs function as his subordinates. The leader is responsible for maintaining the global picture and coordinating the implementation of the team's strategy. *Each subordinate DM is responsible for exercising control of his area of geographic responsibility.* (Song, 1991, p. 1) Figure 1 shows a graphical representation of the command structure.

Each DM sits at a workstation that provides an interactive display of the current tactical situation and allows communication with the other DMs through preformatted messages. The team faces a dynamic environment in which neutrals, decoys and hostile contacts (tasks) arrive at random. Each task must be processed within a finite window of opportunity. The DMs allocate their limited resources to process the tasks according to the information they have gathered. Processing includes detection, identification and prosecution of tasks. The resources are located on platforms "owned" by the subordinate DMs. (Song, 1991, pp. 1-2)
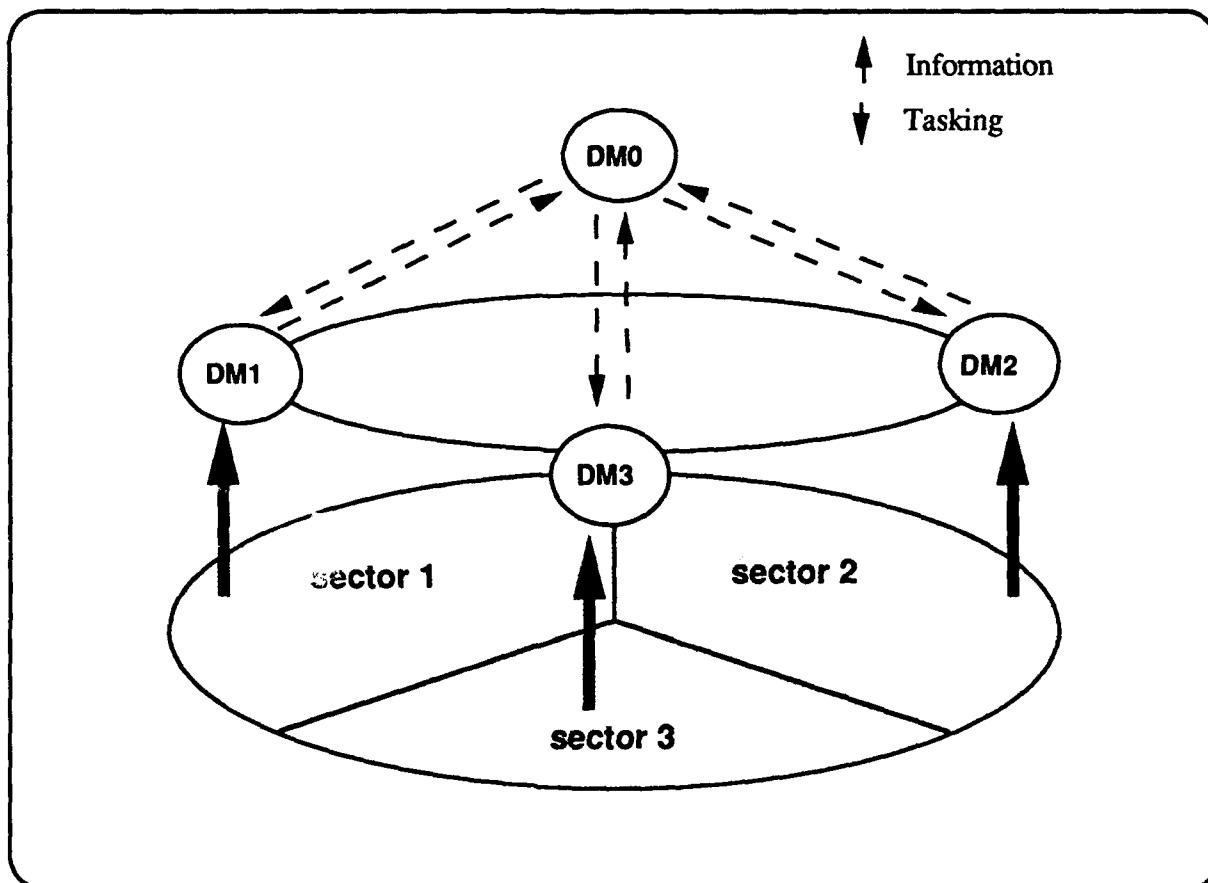
Figure 1   CWC-DDD Command Structure

B. OVERVIEW OF THE CWC-DDD ENVIRONMENT

1. Independent Variables

The RAINCOAT version of the CWC-DDD paradigm is designed to allow manipulation of three independent variables of interest in the study of distributed dynamic decisionmaking. The three are the role of the leader, the information structure, and the level of uncertainty.

a. *Leader's Role in the Team*

The leader's role can be an active role or a passive role. In the active role, the leader has direct control over resource coordination. He can advise a subordinate to transfer platforms to another DM, or he can act unilaterally and force platform transfers among his subordinates. In both cases his actions would be based upon the combination of his assessment of the global situation and the team's overall strategy. (Kleinman and Song, 1990, p. 132)

b. *Information Structure*

The information structure can be centralized, partially centralized, or decentralized. In the centralized information structure case, all DMs will see the same display of task position and attribute measurements. In the decentralized information structure, each subordinate DM will see a display of only those tasks within his area of responsibility. DM0 will see a display combining the displays of the subordinate DMs. In the partially centralized information structure case, it is possible to control what information is displayed to each DM. For example, DM1 may see a display showing the positions and attribute measurements of air tasks only, DM2 may see a display showing the positions and attribute measurements of surface tasks only, and DM3 may see a display showing the positions of subsurface tasks only (Song, 1991, p. 4).

c. Level of Uncertainty

During the game, the DMs identify each task by measuring the attributes of the task. Each task is initially classified as a neutral or a threat. If the task is determined to be a threat, the task is further classified as to threat level. "Sensor noise" is added to the attribute measurements to make identification more difficult. Attribute measurements (with sensor noise) are grouped around a mean in a normal distribution. In the low uncertainty case, the difference between the means of the attribute measurements is large enough to minimize the overlap in the normal distributions caused by the sensor noise. This makes identification easier relative to the high uncertainty case. In the high uncertainty case, the difference between the means of the attribute measurements is smaller and leads to a larger overlap in the normal distributions. Identification in this case is more difficult relative to the low uncertainty case. (Song, 1991, p. 3) Figure 2 is a graphical representation of the normal distributions of the first attribute for neutrals and threats in high and low uncertainty situations encountered in RAINCOAT.

2. Objects Within the Environment

a. Platforms and Subplatforms

Platforms and subplatforms correspond to ships, aircraft and submarines working together as a battle group (Kleinman and Song, 1990, p. 130). The subordinate DMs use the platforms and subplatforms to detect and identify tasks entering their geographic area of responsibility, and to prosecute those tasks potentially threatening their commander at the battle group center.

8
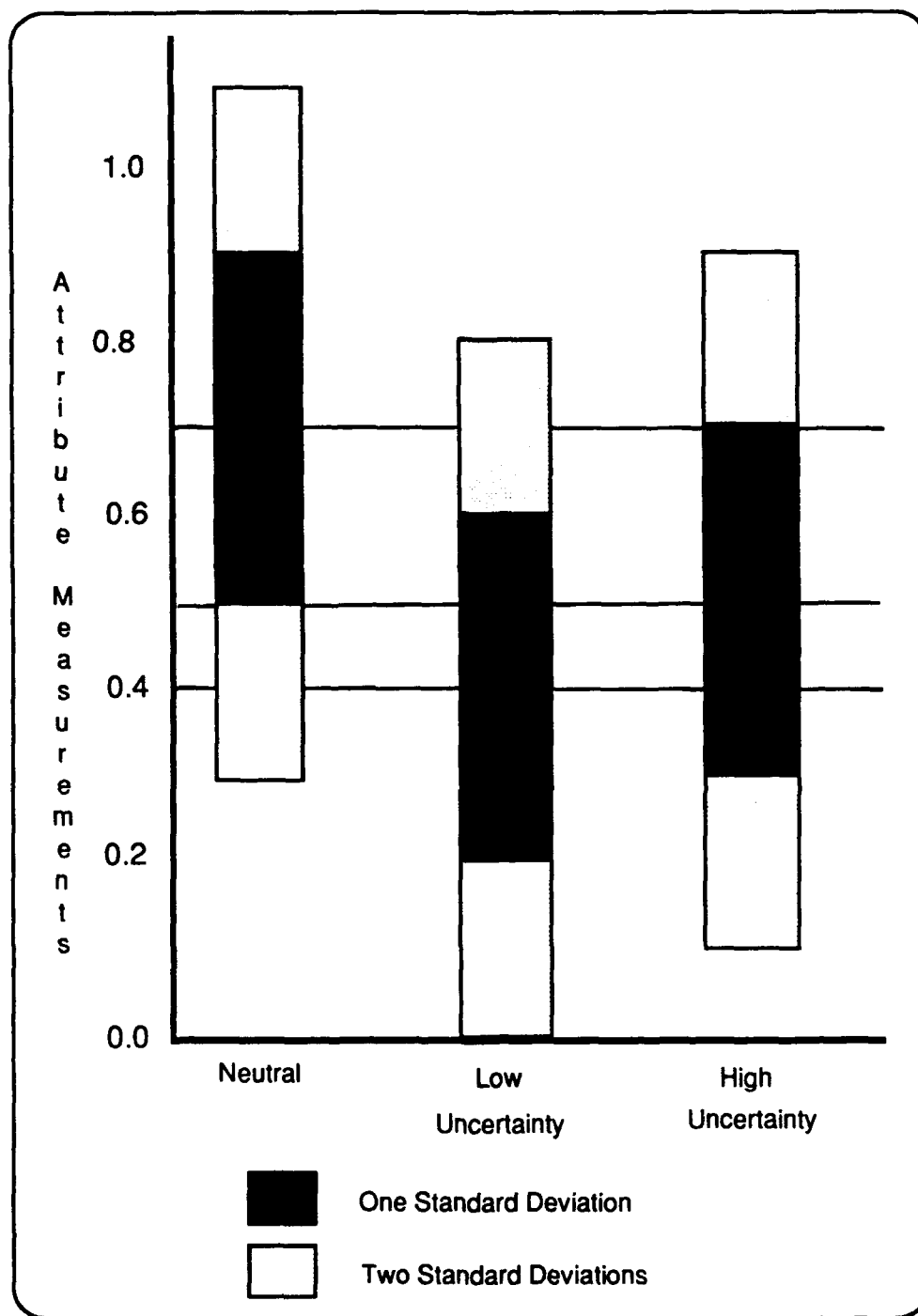
Figure 2   Normal Distribution of the First Attribute

9

(1) Platforms.   Each platform may carry sensors, resources (weapons), and subplatforms. The sensor, weapon and endurance features of each platform class are preprogrammed by the experiment designer.   These parameters do not vary from platform to platform within the same class. However, the experiment designer may vary the number of subplatforms a platform may carry within a platform class. (Kleinman and Song, 1990, p. 130)

(2) Subplatforms. Subplatforms are located on their parent platforms at the beginning of the game. Once the game has begun, each DM may launch one or more subplatforms from his platforms.   Each launched subplatform becomes an independent platform after a preset launch delay. Subplatforms can operate independently of their parent platforms for a limited amount of time. Some subplatform types can return to their parent platforms and be launched again after a "refueling" delay.   This type of subplatform corresponds to reusable items like helicopters. Other subplatform types cannot be returned and simply disappear when their endurance runs out. This type of subplatform corresponds to non-reusable items like sonobuoys. (Wu and Song, 1990, p. 10)

(3) Sensors. Each platform or subplatform may have three types of sensors which provide information on air, surface, and subsurface tasks, respectively. The combination of platform or subplatform class and sensor type determine three ranges for each sensor. The outer sensor range is the detection range. While a task is within this range, the task shows up as a unidentified contact on the DM's display. The middle sensor range is the measurement range.

While the task is within this range, the DM may make a "noisy" measurement of the attributes of the task. The inner sensor range is the classification range. While a task is within this range, the task is classified as to its task class by the sensor. (Wu and Song, 1990, p. 12)

(4) Resources (or weapons). There are three types of resources: air resources, surface resources, and subsurface resources. Each platform and subplatform class has a resource vector that determines the quantity of each resource type onboard platforms of that class. The resource vector also determines the effective range of each resource type for that class. If a platform or subplatform class does not have a particular resource type, platforms or subplatforms of that class cannot attack tasks in that arena. (Wu and Song, 1990, p. 11)

(5) Platform and Subplatform Command and Control. Platforms and subplatforms are controlled by the DM that "owns" them. Move, pursue, attack, and transfer commands are issued through pull down menus. Status information is obtained by double clicking on the platform or subplatform icon to bring up the information window. This information window is also used to display sensor and weapon range rings, and to launch subplatforms. Platforms and subplatforms cannot be owned by more than one DM at a time. They can be transferred from one subordinate DM to another with an attendant time delay. (Kleinman and Song, 1990, p. 130)

b. Tasks

Tasks correspond to contacts that are potential threats to the battle group. Tasks may be air tasks, surface tasks, or subsurface tasks. Tasks appear, maneuver and disappear according to the experiment designer's preprogrammed maneuvers (Kleinman and Song, 1990, p. 130). Tasks do not respond to the actions of the platforms and subplatforms.

(1) Task Classes. Each task type can be further subdivided into classes representing specific subtypes. For example, task type B may represent all air contacts. Task classes BA, BD and BN then may represent MIG-23s, robot decoy drones, and neutral merchant ships, respectively (Kleinman and Song, 1990, p. 130). Decoy tasks do not attack the battle group, although they may mimic a hostile in many respects. Neutral tasks do not pose a threat to the battle group, but their transit path may make them appear to be potential threats. The presence of decoys and neutrals will often cause a DM to tie up resources needed to combat threats elsewhere.

(2) Task Characteristics. Each task has an attribute vector with individual elements that correspond to characteristics of the task. The attribute elements may represent vulnerability, strength, size, etc. The values of the attribute elements for each task are randomly generated from a normal probability distribution using a preprogrammed mean and standard deviation. Each task class attribute element may have a unique mean and standard deviation. The attribute vector is multiplied by the preprogrammed task class matrix to determine the resources required to attack successfully a task. (Wu and Song, 1990, p. 14-15)

12

3. Communication Between DMs

Communication is essential to effective defense of the carrier battle group. Team members are free to share their local information regarding tasks, and to coordinate platform/subplatform ownership and task prosecution. However, verbal exchanges are prohibited. All communication is through preformatted messages. (Kleinman and Song, 1990, p. 130) Figure 3 shows the preformatted messages (Song, 1991, p. 5).

Message	Meaning
Request information	ask another DM to send his information about a task
Request platform	ask another DM to transfer ownership of a platform
Request action	ask another DM to handle / attack a task
Transfer information	transfer the information about a task to another DM
Transfer platform	transfer ownership of platform to another DM
Transfer action	advise another DM that you will handle/attack a task

Figure 3 Preformatted Messages in the Paradigm

Three limitations on communications can be introduced to simulate real world conditions. First, a time delay in message transfer can be introduced to simulate communications and data processing delays. Second, the number of communications in a period can be specified to simulate a limited quantity or limited availability of communications circuits. Third, the structure of the communications network, or who can talk to who, can be specified to simulate the command hierarchy. (Song, 1991, p. 5)

13

4. The Interactive Display

Each DM is provided with an interactive display. The screen is composed of the main display, the status panel, the communication panel, and the prompt panel (Kleinman and Song, 1990, p. 129). A sample screen is shown in Figure 4.

a. Main Display

The main display consists of primarily of a circular display similar to a radar scope. The center of the display represents the position of the carrier. The grey band surrounding the center is the penetration zone (Kleinman and Song, 1990, p. 129). The circles outside this grey band represent range rings. The outermost circle represents the limit of the carrier's detection range. The circular display is divided into twelve 30 degree sectors. Each subordinate DM has responsibility for five of the sectors. This creates a one sector overlap with each of the neighboring DMs.

Platforms, subplatforms, and detected tasks appear on this circular display. The position of each is indicated by an icon. Above each icon is a short alphanumeric descriptor that provides additional information about the object. Platform and subplatform descriptors consist of a number designating the owning DM, a letter indicating the platform or subplatform type, and a three digit number that uniquely identifies this platform or subplatform within its class. Task descriptors consist of a letter designating it as an air, surface, or subsurface task. This is followed by a ? , N, L, M, or H indicating unidentified, neutral, low threat, medium threat, or high threat, respectively. The N, L, M,

14

Figure 4   Sample Interactive Display Screen

15

or H are assigned by the DMs based upon their threat assessments and may or may not accurately reflect the true identity of the task. Figure 5 contains a sample of the icon and alphanumeric designators that may appear.

3:A-004

Type A Platform

A?-219-0

Unidentified Task

1:X-102

Type X Subplatform

AH-228-2

High Threat Task

2:Y-108

Type Y Subplatform

AN-202-3

Neutral Task
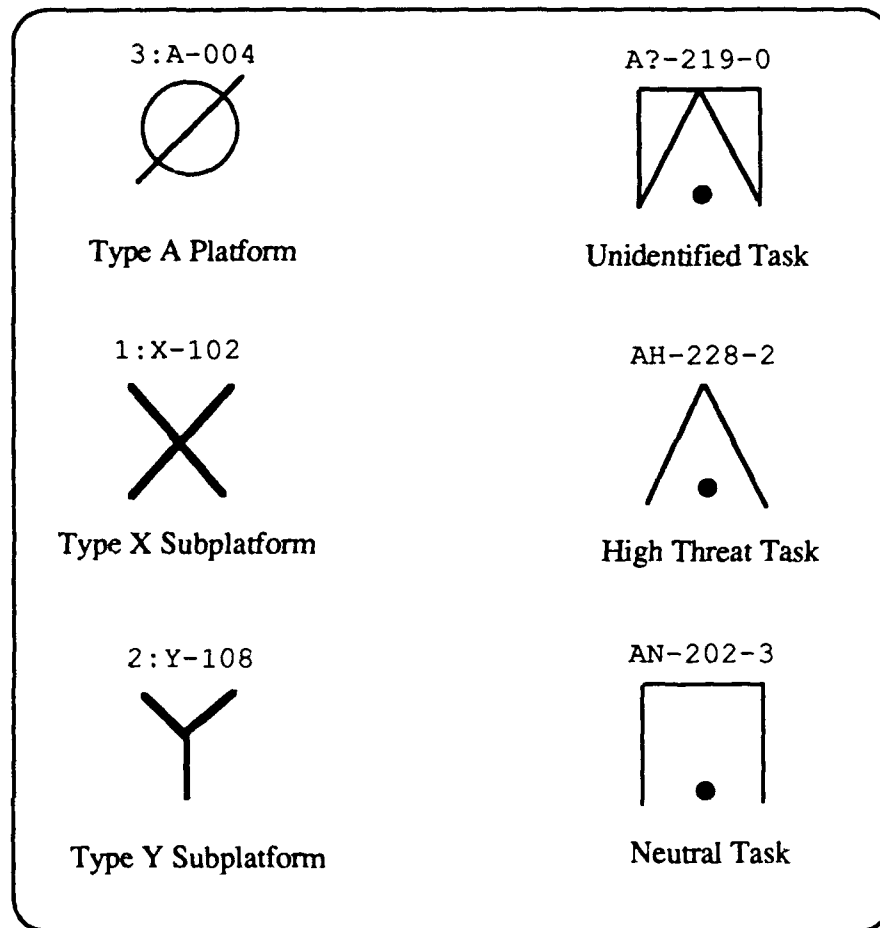
Figure 5   Sample Icons and Alphanumeric Identifiers

Platforms initially appear at predetermined positions around the center of the display. They may be repositioned before the start of the game. Subplatforms are initially on their parent platform. They must be launched from their parent platform after the start of the game to be used. Tasks arrive and move according to a scenario preprogrammed by the experiment designer. Those tasks that are identified as threats should be attacked before the task can reach the penetration (Kleinman and Song, 1990, p. 129). Commands are issued to objects on the main display using pull-down menus, pop-up windows, clicking, or doubleclicking (Kleinman and Song, 1990, p.130).

b. Communications Panel

The communications panel is composed of an incoming and outgoing window. The incoming window displays the messages received from other DMs. These messages include task identification, task coordination, and asset ownership messages. The outgoing window displays feedback information when certain actions are taken or messages sent. This feedback information includes subplatform launch, task coordination, and asset ownership acknowledgements. (Kleinman and Song, 1990, p. 130)

c. Status Panel

The upper portion of the status panel is used to display the current scenario time and team strength. The middle of the status panel contains four buttons. Two of these buttons allow the DMs to zoom in or out as necessary to

get a clearer picture. The bottom of the status window consists of a time to go bar which is used to keep the DMs advised of the status of resource transfers, attacks, or other events that have built in delays. (Kleinman and Song, 1990, p. 130)

d. Prompt Panel

The prompt panel is used primarily to display error messages. These error messages run the gamut from wasted attack to no more subplatforms to be launched.

C. OVERVIEW OF THE CWC-DDD SOFTWARE

The CWC-DDD software consists of four primary parts: Global, Local, User Interface, and Scenario Generator. Global is the runtime communications, control and data processing center. Local is responsible for maintaining the objects and processing commands. User Interface provides the screen displays and accepts the team members' inputs. Scenario Generator is used by the experimental designer to develop the experimental scenarios. (Song, 1991, p. 6) Figure 6 is a graphic depiction of the CWC-DDD architecture (Kleinman and Song, 1990, p. 134).

The system runs as five parallel processes on five workstations. Global runs on one workstation. A copy of local runs on each of the four remaining workstations. All command and control information traffic is carried across an Ethernet using XDR protocol. Timing synchronization and data consistency between the processes are the responsibility of Global. (Kleinman and Song, 1990, p. 134)
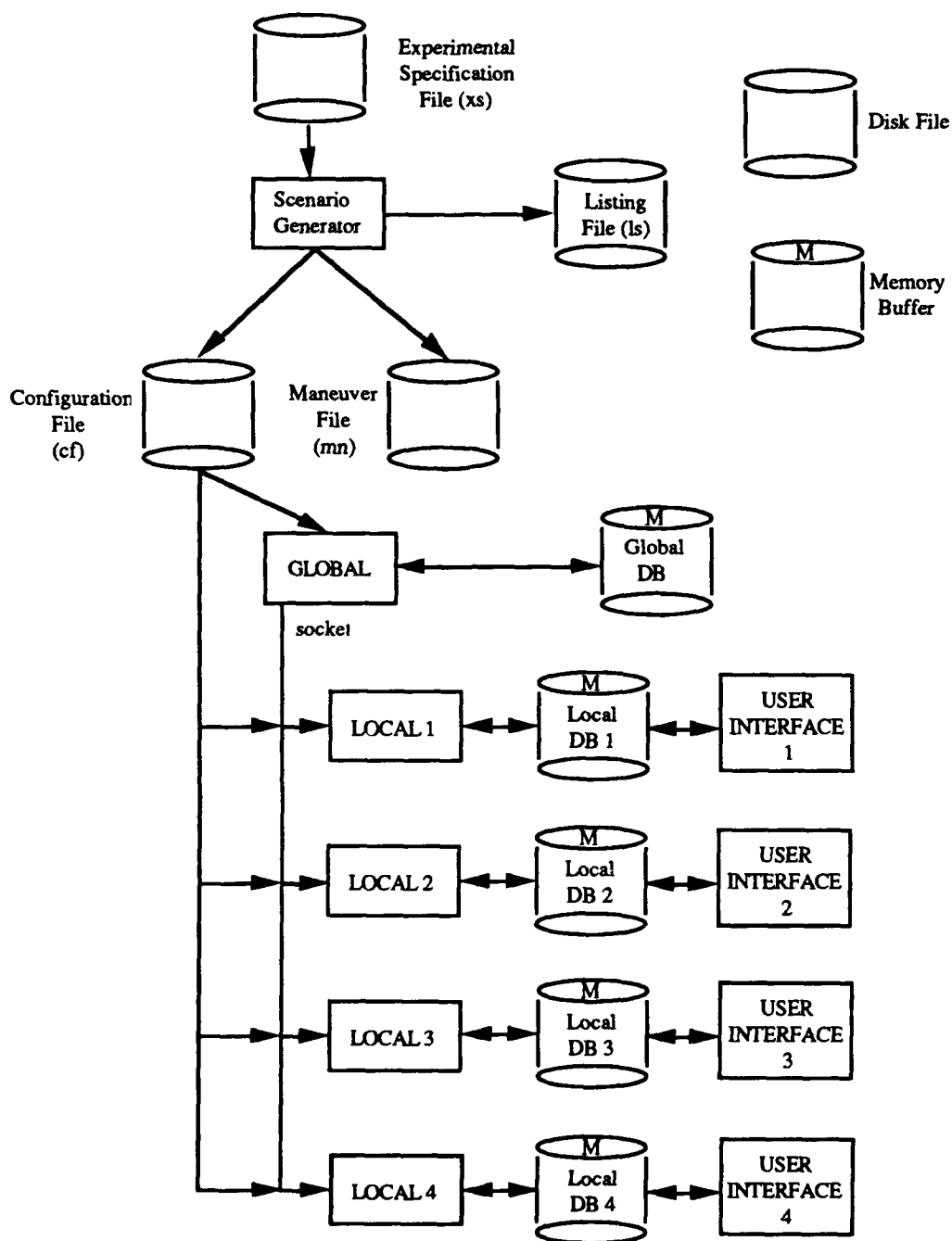
Figure 6   CWC-DDD Composite Structure

The system was written in the C programming language. The Global, Local and User Interface consist of about 30,000 lines of the code (Song, 1991, p. 6). The Scenario Generator consists of about 5,000 additional lines of code (Song, 1991, p. 22). The system utilizes the Sunview windowing system. The current implementation of the paradigm is not portable to other Unix windowing environments.

1. Global

a. Notifier-Based System

Global is a notification-based system. Procedures are registered with the global notifier upon initialization. The global calls the appropriate procedure based upon the socket input received from the locals and the timer pulses received.

A notification-based system is different from the normal main control loop of conventional programming. In conventional programming, the main control loop resides in the application. The main control loop reads the inputs sequentially, acts based upon these inputs, waits for time to expire, and then begins the loop again. If no inputs are received, the main control loop will continue to loop. The conventional main control loop is more appropriate for process driven environments with fewer events or inputs.

In a notification-based system, the main control loop resides in the notifier not the application. The notifier reads the events and "notifies" the appropriate previously registered procedures based upon the events received. If no events are received, the notifier will wait passively for an event to occur. The notification-based system is more appropriate in complex event driven environments.

b. Global Control Flow

Global has two handlers. The first handler is for the timer pulse input and the second handler is for inputs from the Locals. Both handlers are registered with the notifier before starting the real time loop. Once the real time loop is started, the notifier will receive all incoming messages or events and call the appropriate handler to process this message. When a timer pulse is received, the notifier calls the time_event processing procedure. This procedure first sends the time to each Local, and then sends event information and actions to the appropriate Local. When an input is received from a Local, the notifier calls the local_input processing procedure to process the inputs from the locals. The control flow of Global is shown in Figure 7 (Song, 1991, p. 13).

The Global acts as a clearinghouse for the system. It receives and processes all the inputs from the Locals. In response to these inputs, it generates messages to be sent to the Locals for action. Those messages that are to be acted upon without delay are placed in a buffer. Those messages that are to be executed after a time delay (i.e. communications delay) are placed in a linked list . When the notifier receives the timer signal, it first sends the time to all the Locals and then sends any messages in the buffer to the appropriate Local. (Song, 1991, p. 12) The information flow of Global is shown in Figure 8 (Song, 1991, p. 14).

c. Real Time Control

The CWC-DDD paradigm is a real time simulation. The system time is kept in Global. At a fixed time interval, a timer pulse is sent to the notifier, which in turn sends the time on to each of the Locals. (Song, 1991, p. 15)

21

Timer Pulse   Input from Locals

NOTIFIER

process time_event   process local_input

advance
and send
time

pick up
current
event

send
message
to
locals

process
move
command

. . . .

process
evade
command

process various commands
according to message received
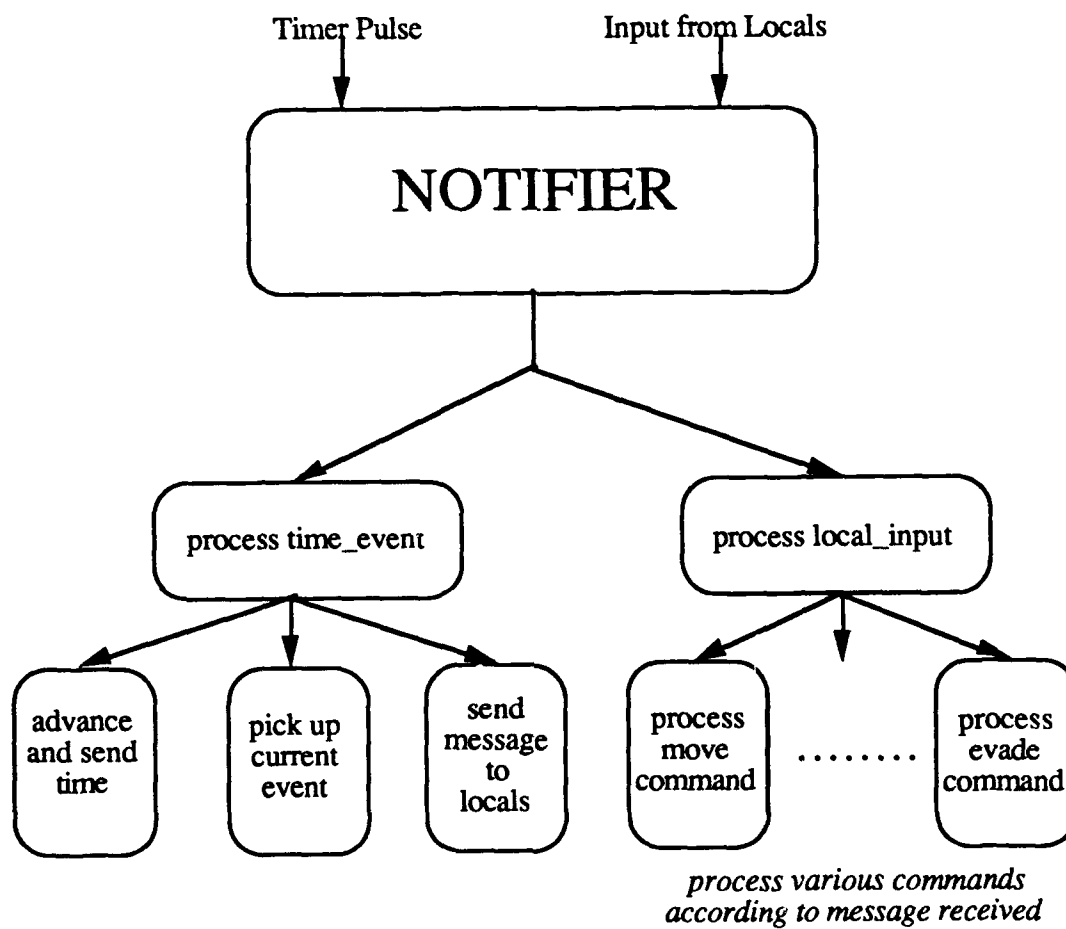
Figure 7   Control Flow of Global

22
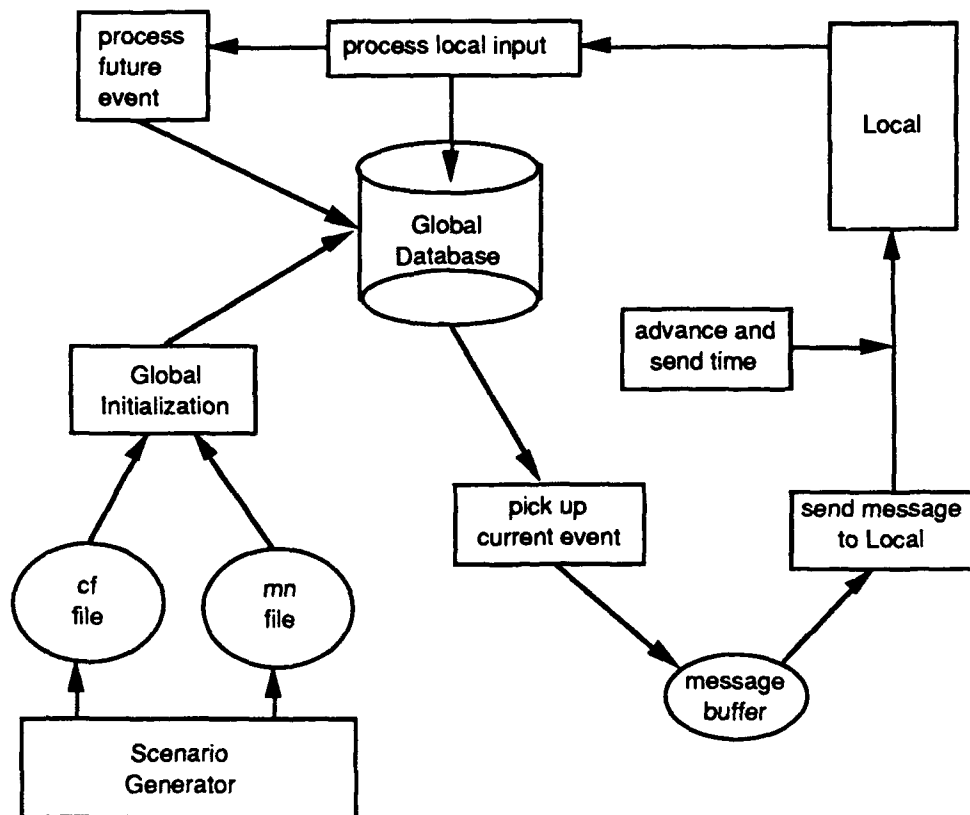
Figure 8   Information Flow of Global

In general, events in the real world do not happen instantaneously, but occur after a certain time delay. In order to simulate this time delay, Global maintains a future event linked list. This linked list is ordered by time of event occurrence. As new events are generated, they are added to this linked list. At a fixed time interval, Global checks the linked list and sends the events that should be executed to the appropriate Local for action. When an event is sent to a Local for action, it is deleted from the linked list. Additionally, the Global will delete a future event from the linked list if a new input from a Local will cause the event to no longer to occur. For example, when a task is destroyed all future events for this task are removed from the linked list. (Song, 1991, p. 15)

The CWC-DDD paradigm actually maintains two future event linked lists in Global. One linked list is for future asset events. It is initialized by reading the configuration file. It is changed by the inputs received by Global from the Locals. The other linked list is for future task events. It is initialized by reading the maneuver file. It is changed only when a task is destroyed. (Song, 1991, p. 15)

d. Data Consistency

The CWC-DDD paradigm uses a pseudo-distributed database. Each Local receives a copy of the database at the beginning of the scenario. The Locals update and change the data in their database based upon the actions of their DM and messages received from Global. Every change or update made to one Local database must be sent to the other Local databases so that their information may be updated or changed also. Data inconsistency occurs when two or more Locals attempt to change the same data in their database at the same time. (Song, 1991, p. 16)

24

Global is tasked with maintaining certain central information lists to overcome the potential problem of data inconsistency. These central lists contain flags that are set when a related data item is being changed by a Local. Every update or change to the database is sent to Global. Global checks its central lists to see if this data item can be changed. If it can be changed, the Global sends the change or update to all the Locals. If it cannot be changed, Global refuses the update or change. The originating Local does not update its database until it receives the return message from Global. (Song, 1991, p. 16)

e. Data Collection and Analysis

Global is also responsible for recording all properly executed commands in a log file. At the end of each scenario, Global sorts and analyzes the log file. The results of this analysis are output as an experimental report. The log file may also be used to replay the scenario. (Song, 1991, p. 17)

2. Local

a. Function of Local

Local is responsible for processing the commands of its DM. DMs issue commands using pull down menus and mouse clicks. The menu selections and mouse clicks are read by the User Interface and passed to the Local for processing. If the command is properly issued, the Local will process it and pass it to the other Locals through Global. If the command is improperly issued, the Local will have the User Interface display an error message in the prompt window. (Song, 1991, p. 23)

Local also controls the display of the assets and tasks on the screen. Before the start of the scenario, the DMs are allowed to position their platforms in accordance with their strategy. These initial positions are passed to Global by

Local. Once the scenario has begun, as Local receives the time from Global it updates the positions of the assets and tasks using the object's velocity and the time passed. The new positions are then passed to User Interface for display of the appropriate icon. (Song, 1991, p. 23)

b. Structure of Local

Local, like Global, is notifier-based. When a DM enters a command, the workstation's notifier calls User Interface to get the input. User Interface in turn calls process_command to take whatever action is required in response to the command. When Local receives a message from Global, the notifier of the workstation calls process_global_message to process the message. (Song, 1991, p. 24) The structure of Local is shown in Figure 9 (Song, 1991, p. 25).

Each Local consists of four modules: initialization, get_db, process_global_message, and process_command. The initialization module is responsible for registering the notifier handlers, making the socket connections with Global, creating the Local database, and creating and initializing the display. The get_db module is responsible for supplying the most current information about objects from the Local database to User Interface. The process_global_message module is called by the notifier to process the messages received from Global. The process_command module is responsible for taking the command information from User Interface, taking the appropriate action, and sending a message to Global informing it what has happened and asking for further instructions. (Song, 1991, p. 24)
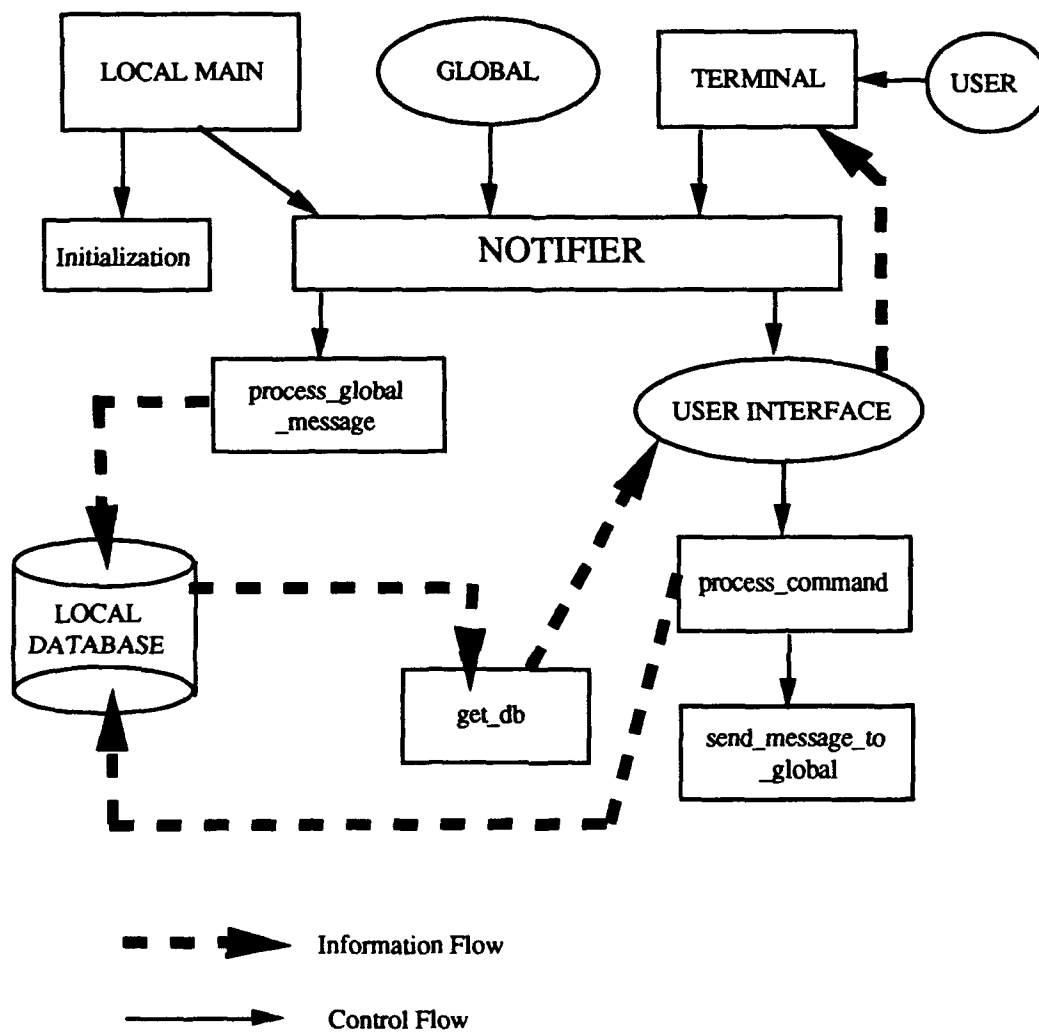
Figure 9 Local Structure

c. Command Processing By Local

Every command is processed in a two step process. When the Local receives a command, the process_command module is called to do the appropriate calculations. Information about the command and the results of the calculations are then sent to Global for approval. The Local database remains unchanged at this point. When Local receives authorization from Global through the process_global_message module, it completes execution of the command. Local now updates the database with the information in the message from Global. (Song, 1991, p. 26)

3. User Interface

The User Interface has two major responsibilities. The first is to keep the screen up to date with the progress of the scenario being played and to display communication messages to the player. This is accomplished by consulting the Local database during every time period and updating any screen information that might have changed. The second responsibility is to allow the player to enter commands that will alter the course of the scenario. This is accomplished by processing the user input and sending the appropriate message to the Local notifier. (Song, 1991, p. 28)

4. Scenario Generator

The main function of the Scenario Generator is to assist the experiment designer in developing a scenario (Song, 1991, p. 18). It is used to configure the assets, to define the task attributes, to design the task movements, and to specify

the environment the scenario will be run in. All of the experimental parameters have default values which provide the designer with a reasonable set of initial parameters. The designer need only specify those parameters that are unique to his experiment. (Kleinman and Song, 1990, p. 132)

The interface between the Scenario Generator and the experimental designer is a flexible experiment description language - XS language. Each XS language statement consists of keywords and the associated values of the parameters. Comments may be added to the XS source file to make the file even more readable. (Kleinman and Song, 1990, p. 133)

The generation of a scenario is a two step process. First, the experimental designer creates the XS source file which specifies the parameters that are unique to his experiment. Second, this file is used as the input to the Scenario Generator. The output is: 1) a MN file which contains all the task maneuvers in chronological order; 2) a CF file which contains environmental as well as the attributes of the assets and the tasks; and 3) a LS file that lists all the experimental parameters, including the defaults that were used. If needed, either the XS or LS file may be modified and used as the basis for another scenario. (Song, 1991, p. 19)

III. IMPROVING THE CWC-DDD PARADIGM

A. REALISM OF THE CWC-DDD PARADIGM

The CWC-DDD paradigm was designed to simulate "real world" Naval engagement scenarios. However, the "real world" is a complex place with numerous environmental variables. Implementation of too many environmental variables could greatly detract from the experimenter's ability to control the experiments and produce meaningful results. Implementation of too few environmental variables would result in a controllable experiment, however, if the realism of the model is limited, the results will not be transferable to real world situations. As with most models, a middle ground must be selected that represents a compromise between controllability and realism. Does the current version of the CWC-DDD paradigm represent an acceptable middle ground?

During August and September 1991, a group of 28 military officers were the subject of multiple trials using the RAINCOAT version of the CWC-DDD paradigm. At the conclusion of their trials, each subject was asked to provide some brief comments on the realism of the paradigm based upon their field experience. The responses collected were primarily anecdotal. The majority did agree that the paradigm was a reasonable approximation the real world. When asked for specific ways to improve the paradigm, most felt that the tasks (enemy) needed to be responsive to the actions of the assets and the environment.

B. ADDING "INTELLIGENCE" TO THE CWC-DDD PARADIGM

Assets act as directed by their controlling player. Environmental conditions change as the scenario progresses. There is no way to predict in advance how a player will use his assets, or when a given set of environmental conditions will exist. Yet the task must have the ability to evaluate the current situation and select the correct response to the situation. The task must have a "rudimentary intelligence" built into it to allow it to select the correct response.

One common way of adding intelligence is to develop a set of heuristics that can be hard coded into the system. The heuristics are generally of the type "if x, y and z are true, then do action A." The more complete the set of heuristics the more "intelligent" the system component will seem.

For the CWC-DDD paradigm, a trial group of five heuristics were developed. The conditions and the response for each heuristic were carefully crafted so as to duplicate as closely as possible the real world conditions and response. The set of heuristics was reviewed by several military officers for adherence to generally accepted battle procedures prior to implementation. The code for implementing the heuristics can be found in the appendix.

C. THE WEAPONS RANGE HEURISTIC

1. The Heuristic

If a threat comes within the weapons range of any asset, then the threat shall change course at a random angle away from the asset.

2. Implementation of the Heuristic

a. Determining If a Task Is Within the Weapons Range of Any Asset

In order to determine if a task is within the weapons range of an asset, a four step evasion determination process is executed within Local. First, the task type is determined to be air, surface, or subsurface. Second, the asset's weapons range against that type of task is determined. Third, the distance between the task and the asset is calculated using the distance formula. Fourth, the distance and the weapons range are compared. If the distance is less than the weapons range, a function to generate an evasive maneuver (within_weapons_range) is called. This evasion determination process is repeated for every task and asset pair.

The first three steps of the evasion determination process were previously implemented as a for loop in the update_task_state function in the "unintelligent" version of the paradigm. However, to properly implement the heuristic, three additional checks must be made by the loop before within_weapons_range is called to generate an evasive maneuver.

In the real world, only threats would attempt to evade assets. Neutral tasks would maintain their course and speed. The evasion determination process above would cause evasive maneuvers to be generated for both threats and neutrals. Neutral tasks are eliminated from consideration by using the is_threat function to determine if a task is a threat. If the task is not a threat, the distance between the task and the asset is not calculated and within_weapons_range is not called for this task.

During execution of the paradigm, there are four Locals running. Under the evasion determination process above, each Local would generate its own evasive maneuver for each threat within an asset's weapons range. To prevent this, DM0's Local is given responsibility for determining if an evasive maneuver is required and calling within_weapons_range to calculate it. The other Locals will encounter an if statement that will cause them to skip the evasion determination process.

The function update_task_state is called once a second for each task. If allowed to run in this manner, within_weapons_range would calculate an evasive maneuver for each threat within an asset's weapons range once a second. The generation and sending of that many evasive maneuvers to Global quickly overloads the network communications pathways and causes the paradigm to abort execution. Additionally, it is unrealistic to expect an enemy task commander to attempt a new evasive maneuver every second. It is more likely that this enemy task commander will attempt one evasive maneuver and wait a period of time to see if it will succeed.

These two problems are overcome by having the threat execute one evasive maneuver when it first enters an asset's weapons range. The threat will not execute another evasive maneuver relative to that asset until it moves outside that asset's weapons range and back in again. It will execute a second evasive maneuver though, if it comes within the weapons range of another asset. This reduces the number of evasive maneuvers being sent on the network to a manageable number, and also adds the realism of an enemy task commander waiting a period of time to see if his evasive maneuver will succeed.

b. Calculating the Evasive Maneuver

The function within_weapons_range calculates the evasive maneuver for the threat based upon the threat's position, the asset's position, and the distance between them. This evasive maneuver is not immediately implemented by the Local, but is sent to Global, and only implemented when Global sends it back to each Local. The maneuver is routed through Global so that all the Locals will receive the information and update their databases at the same time.

Initially, within_weapons_range checks to see if the threat is within the inner radius. The inner radius represents the range at which an asset's attack cannot be completed prior to a threat being able to penetrate the penetration zone. If a threat is at or inside the inner radius, the threat does not make any type of evasive maneuver, but instead continues towards the penetration zone.

Next, within_weapons_range calculates the angle, the distance, and the time for the evasive maneuver. The angle is calculated by taking the angle formed by the positive x axis and a line connecting the threat and asset positions and adding to it a random angle between -90° and 90°. The distance is calculated by multiplying the distance between the threat and asset positions by a random number between 0 and 1. The time is calculated by dividing the distance for the evasive maneuver by the maximum speed for this task. Figure 10 is a graphical representation showing the relationships between the different quantities.

Within_weapons_range now sends the evasive maneuver to Global in the form of a task_evade_msg. The task_evade_msg contains:

- the threat's id,
- the maximum speed of the task,
- the x and y coordinates of the threat's position,
- the x and y components of the threat's velocity during the evasive maneuver,
- the x and y coordinates of the endpoint of the evasive maneuver,
- the time required to accomplish the evasive maneuver.

c. Global's Response

When Global receives a task_evade_msg from Local, it immediately writes the message to the log file and places the message in the buffer. At the end of the next time period Global will send the contents of the buffer to each of the Locals for action.

Global will also create a TASK_CHANGE event to be added to the task future events linked list. The TASK_CHANGE event will be created so that the threat is directed towards the next scheduled TASK_CHANGE event for this threat. Global will check the velocity required to reach this next TASK_CHANGE event point on time. If the velocity exceeds the maximum velocity of the threat, it will adjust the event times of all future events to reflect the additional time required for the threat to reach the future event points.

D. THE SECTOR CHANGE HEURISTIC

1. The Heuristic

If the number of tasks is exceeded by the number of assets within one of the twelve sectors, then the tasks within that sector shall change course for the neighboring sector with the fewest assets.
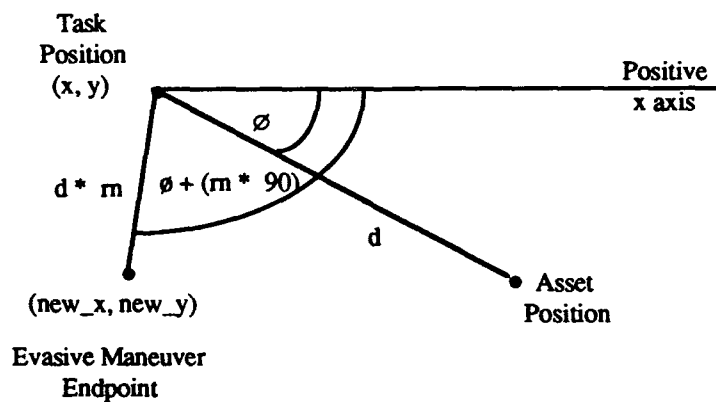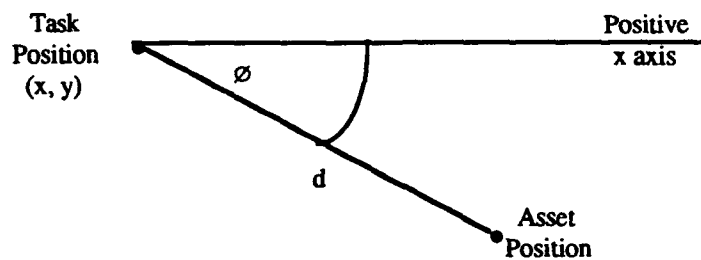
ø = angle between line connecting task and asset positions and horizontal x axis
d = distance between task and asset positions
m = random number
d * m = random distance for evasive maneuver
ø + (m * 90) = random angle for evasive maneuver
new_x = x coordinate of maneuver endpoint
new_y = y coordinate of maneuver endpoint

Figure 10   Relationship Between Evasive Maneuver Quantities

2. Implementation of the Heuristic

a. *The sector_check Function*

The sector_check function checks if the number of tasks within a sector is exceeded by the number of assets. It is called by Local as part of the update function. However, during the execution of the paradigm there are four Locals running. To prevent each Local from performing the sector_check function, DM0 is given responsibility for performing the sector checking. The other Locals will encounter an if statement that will cause them to bypass execution of sector_check.

b. *Maintaining the Asset and Task Counts*

The sector_check function requires a count of the number of assets and tasks in each of the twelve sectors. Realistically, these counts would be maintained by command and control (C2) task located just outside the detection range of the central asset (the C2 asset). The detection range for this C2 task would not be unlimited, but would be the same as the central asset's detection range. The C2 task's detection range will prevent it from obtaining accurate counts in all sectors.

For purposes of the sector_check function, the C2 task is located at a point on the primary attack axis just outside the central asset's detection range. This allows the C2 task to obtain maximum coverage of the primary attack and surrounding sectors. Coverage is sacrificed in the area opposite the primary attack axis. Figure 11 shows the relationship between the detection ranges.

37
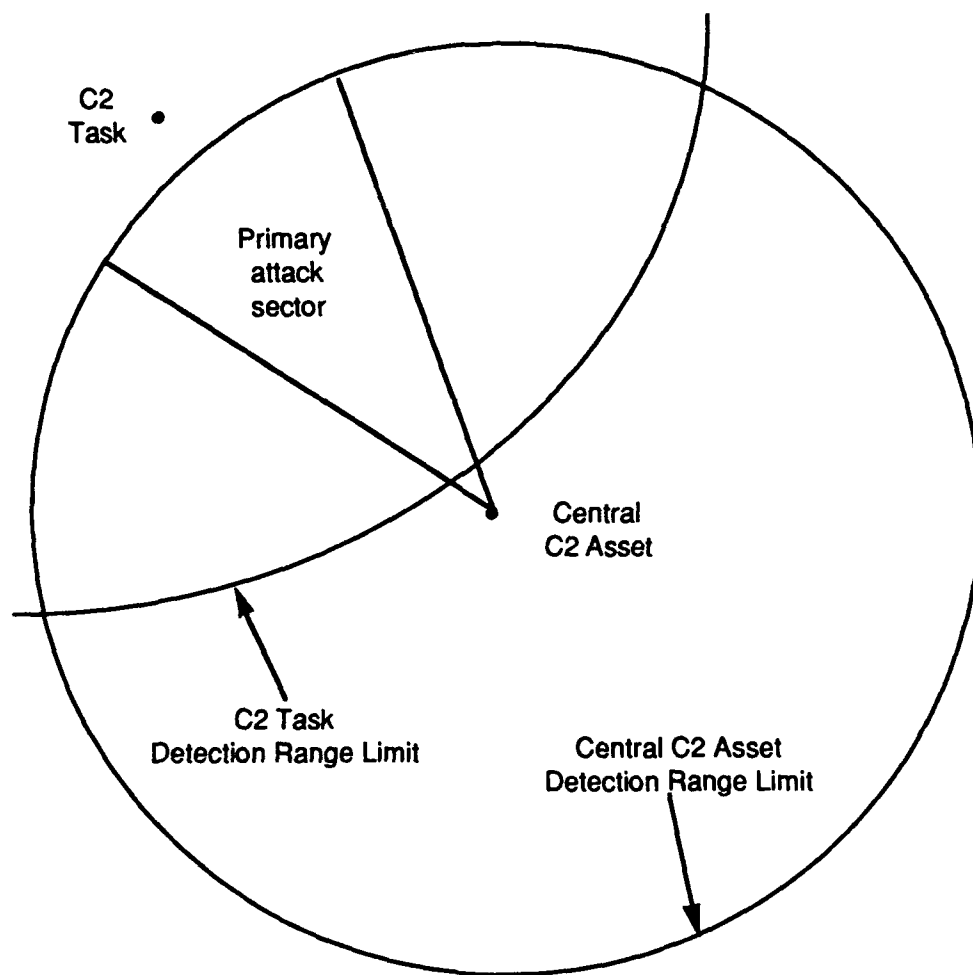
Figure 11   Detection Ranges of C2 Asset and C2 Task

Counting the assets and tasks is accomplished by a simple three step process. First, two arrays of 12 values each are created and initialized to zero. Second, the distance between the C2 task position and the position of each asset is calculated and compared to the detection range of the C2 task. If the C2 task can "see" the asset, one is added to the asset count for that sector. The central asset is ignored in this count since it is not in any sector. Third, the distance calculation and comparison are done for each task. If the C2 task can "see" the task, one is added to the task count for that sector.

c. Calculating the Evasive Maneuver

Calculating an evasive maneuver for a each evading task requires the selection of a destination point in the appropriate adjoining sector. This is accomplished by calculating the distance between the task's position and the center of the display. This distance is then multiplied by a random number between 0 and 1 to get the distance the destination point will be from the center. The destination point is a point at this "random" distance from the center on a line that bisects the appropriate adjoining sector. Figure 12 shows a graphical depiction of the relationship between the task position and destination point.

Using this destination point and the task's position, the velocity and the time required to move from the task's current position to the destination point are calculated and sent to Global in the form of a task_evade_msg. This process is repeated for each task requiring an evasive maneuver.

d. Global's Response

When Global receives a task_evade_msg from Local, it immediately writes the message to the log file and places the message in the buffer. At the end of the next time period Global will send the contents of the buffer to each of the Locals for action.

Global will also create a TASK_CHANGE event to be added to the task future events linked list. The TASK_CHANGE event will be created so that the threat is directed towards the next scheduled TASK_CHANGE event for this threat. Global will check the velocity required to reach this next TASK_CHANGE event point on time. If the velocity exceeds the maximum velocity of the threat, it will adjust the event times of all future events to reflect the additional time required for the threat to reach the future event points.

Figure 12 Relationship of Task Position and Destination Point

E. RESOURCES ENROUTE HEURISTIC

1. The Heuristic

If the courses of assets with sufficient resources to destroy a threat show convergence on the threat, the threat shall change course at a random angle.

2. Implementation of the Heuristic

a. The resources_on_way Function

The resources_on_way function determines if the resources enroute to a threat are greater than or equal to the resources required to properly destroy the threat. If they are, the function generates an evasive maneuver for the threat.

During execution of the paradigm, there are four Locals running. Without intervention, each Local would generate its own evasive maneuver for each threat being faced with sufficient resources to destroy it. To prevent this, DM1's Local is given responsibility for determining if an evasive maneuver is required and calculating it. The other Locals will encounter an if statement that will cause the call to resources_on_way to be skipped.

b. Determining the Resources Enroute a Threat

The Local first determines which assets are enroute towards a threat. This is accomplished by calculating the slope of the asset's velocity vector. This slope is compared to the slope of a directed line segment connecting the asset and task positions. The slopes are considered equal if they are within plus or minus 5% of one another.

Comparing the slopes is not sufficient to determine if an asset is enroute. Since the slopes being compared are the slopes of a vector and a directed line segment, the signs of the x and y components must also be compared. If the signs of the components are not compared, it is conceivable that an asset could be determined to be enroute to a threat when in fact its velocity is directing it on a course directly away from the threat rather then towards the threat. Figure 13 illustrates this possibility. If the slopes and the signs of the components are equal, the asset is considered enroute to the threat.
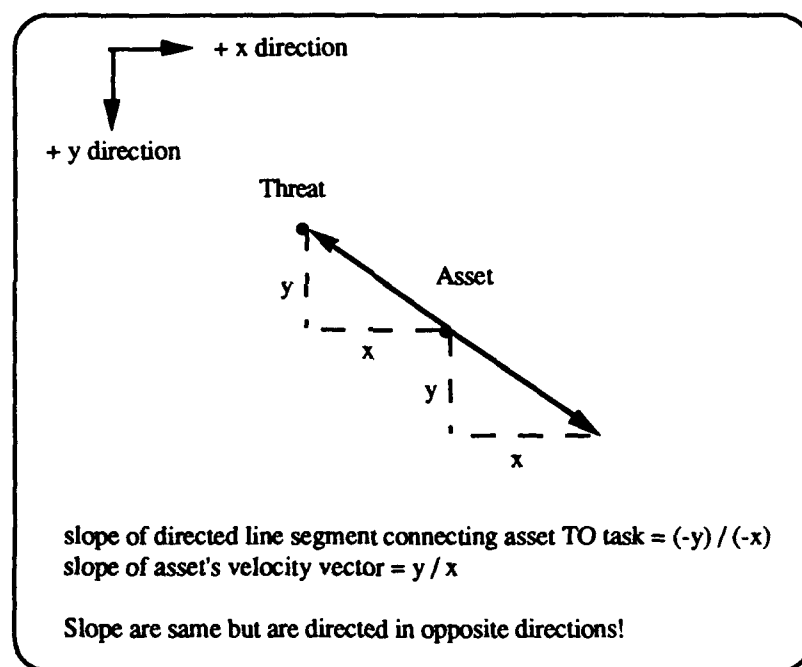
Figure 13 Illustration of Problem With Slope Comparison

Once it has been determined that an asset is enroute to a threat, the amount of resources carried by that asset must be calculated. Each asset has an array of three values that represent the strength coefficients for the weapons the

asset carries. Each asset class has an array of three values that represent the strength of the weapons carried by that class of assets. Each asset can also be designated to represent more than one real world asset (i.e. one asset in the scenario represents a flight of 12 planes) each with their own weapons. The resources enroute with that asset is then equal to the asset's strength coefficient matrix multiplied by the class' weapons matrix multiplied by the number of real world assets the asset represents. The total resources enroute is the sum of the resources enroute for each of the assets enroute to the threat.

c. Determining If the Resources Enroute Are Sufficient

Each task has a resources required array associated with it. Each of the three values in this array represent the resources of that type of weapon required to properly destroy the task. Direct comparison of the elements of the resources with the total resources enroute will determine if sufficient resources are enroute.

In the RAINCOAT version of the paradigm the resources required to properly destroy a threat is not based on the resources required array, but on the threat class. A threat class of 1, requires 1 resource to properly destroy; a threat class of 2 requires 2 resources; a threat class of 3 requires 3 resources. Comparing this number with the first element of the total resources enroute array will determine if sufficient resources are enroute. This RAINCOAT method of comparing the resources available and the resources required was the one implemented.

d. Calculating the Evasive Maneuver

The evasive maneuver is calculated in three steps. The angle for the maneuver is calculated by adding a random angle between - 90° and 90° to

43

the angle the threat's velocity vector makes with the positive x axis. The distance for the maneuver is a random number between 0 and 0.05. The time to accomplish the maneuver is calculated by dividing the maneuver distance by the maximum speed for this threat. This information is sent to Global in the form of a task_evade_msg.

e. Global's Response

When Global receives a task_evade_msg from Local, it immediately writes the message to the log file and places the message in the buffer. At the end of the next time period Global will send the contents of the buffer to each of the Locals for action.

Global will also create a TASK_CHANGE event to be added to the task future events linked list. The TASK_CHANGE event will be created so that the threat is directed towards the next scheduled TASK_CHANGE event for this threat. Global will check the velocity required to reach this next TASK_CHANGE event point on time. If the velocity exceeds the maximum velocity of the threat, it will adjust the event times of all future events to reflect the additional time required for the threat to reach the future event points.
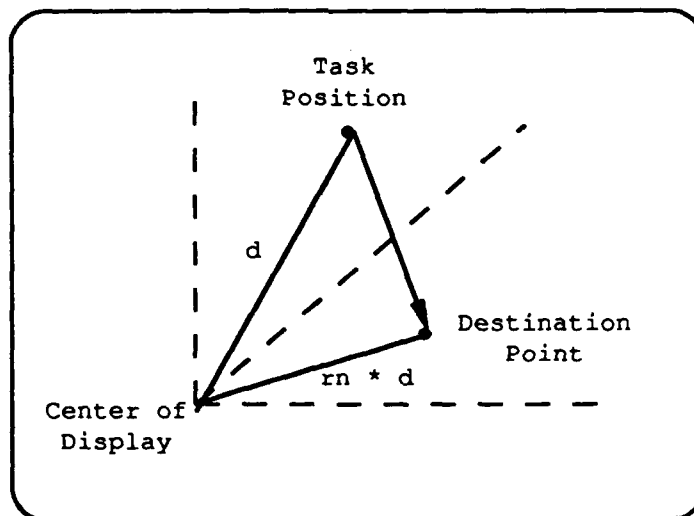
F. PART OF PRIMARY ATTACK HEURISTIC

1. The Heuristic

If a threat is part of the primary attack and that threat succeeds in reaching the first range ring outside the penetration zone (the range at which diversionary attackers normally veer off), then the threat shall change course directly for the penetration zone and accelerate to maximum speed.

2. Implementation of the Heuristic

The attempt_penetration function determines if a threat is in the primary attack, and if the threat is within the inner radius. If the threat meets both of these conditions, this function calculates the changes required to put the threat on a course directly for the center of the penetration zone at maximum speed. This message is sent to Global in the form of a task_change_dir_msg.

Threats do not have a variable associated with them that will identify them as part of the primary attack. However, all threats that are part of the primary attack will center their attack around the primary attack axis. Additionally, all threats that are not part of the primary attack will veer at or before reaching the inner radius. One can reasonably assume then that any threat on the primary attack axis and within the inner radius is part of the primary attack.

When Global receives a task_change_dir_msg from Local, it immediately writes the message to the log file and places the message in the buffer. At the end of the next time period Global will send the contents of the buffer to each of the Locals for action. Global will also remove all future events for this threat from the task future event list.

During execution of the paradigm, there are four Locals running. Without intervention, each Local would generate its own change maneuver for each threat that was part of the primary attack and within the inner radius. To prevent this, DM1's Local is given responsibility for determining if an evasive maneuver is required and calculating it. The other Locals will encounter an if statement that will cause the call to resources_on_way to be skipped.

The attempt_penetration function is part of the update function. Update is executed once a second, however, there is no need for attempt_penetration to be executed once a second. It must be executed often enough to change the courses of the appropriate threats quickly, but not so often as to overburden the network with unnecessary task_change_dir_msg. For this reason, a simple counter loop has been implemented so that attempt_penetration is executed once every five seconds.

G. PART OF DIVERSIONARY ATTACK HEURISTIC

1. The Heuristic

If a task is part of the diversionary attack and that task succeeds in getting closer to the penetration zone than any asset other than the central asset, then that task shall proceed to attempt to penetrate the penetration zone rather than veering off.

2. Implementation of the Heuristic

The change_to_attacker function determines if a threat is in the diversionary attack, and if the threat is inside all assets except for the central asset. If the threat meets both of these conditions, this function calculates the changes required to put the threat on a course directly for the center of the penetration zone at maximum speed. This message is sent to Global in the form of a task_change_dir_msg.

Threats do not have a variable associated with them that will identify them as part of the diversionary attack. This function assumes that any threat that is inside all the assets and is in the diversionary attack axis is part of the diversionary attack.

46

When Global receives a task_change_dir_msg from Local, it immediately writes the message to the log file and places the message in the buffer. At the end of the next time period Global will send the contents of the buffer to each of the Locals for action. Global will also remove all future events for this threat from the task future event list.

During execution of the paradigm, there are four Locals running. Without intervention, each Local would generate its own change maneuver for each threat that was part of the diversionary attack and inside all the assets. To prevent this, DM1's Local is given responsibility for determining if an evasive maneuver is required and calculating it. The other Locals will encounter an if statement that will cause the call to resources_on_way to be skipped.

The change_to_attacker function is part of the update function. Update is executed once a second, however, there is no need for change_to_attacker to be executed once a second. It must be executed often enough to change the courses of the appropriate threats quickly, but not so often as to overburden the network with unnecessary task_change_dir_msg. For this reason, a simple counter loop has been implemented so that change_to_attacker is executed once every five seconds.

H. VALIDATION AND TESTING OF THE HEURISTICS

1. Unit Testing

Each heuristic module was subjected to two types of unit testing prior to its integration with the other heuristic models. The first test consisted of running the modified paradigm and treating the heuristic module as a black box. The inputs to this black box were a contrived set of actions by the DMs that would cause the module to execute. The outputs expected from this black box were the

47

execution of evasive or course change maneuvers by the threats. For example, with the weapons range heuristic DMs continually moved their subplatforms so that the threats came within the weapons range of the subplatform. If the threat executed an evasive maneuver, the test was considered successful.

The second test consisted of treating the heuristic module as a white box. Print statements were placed before and after blocks of code within the module. These print statements were used to print out the values of variables under study or simply to indicate that a logical path in the code had been executed. The variables values printed out before the blocks of code were executed were used to predict the values of these variables after the code blocks were executed. The logical path print statements were used to check the conditions under which different code blocks were executed. The tests were considered successful if the output values matched the values predicted from the input values, and if the correct logical path was chosen for the input conditions.

2. Integration Testing

Integration testing consisted of running the modified CWC-DDD paradigm under experimental conditions with students from the Joint Command, Control and Communications curriculum as subjects. The subjects were instructed to respond to the paradigm normally, but to report any anomalies in the actions of the task if they occurred. At the conclusion of each run, the subjects were questioned about the realism of the task actions. The tests were considered a success if no anomalies were reported and the subjects felt the actions were reasonably realistic.

48

3. Results of Testing

Extensive unit testing of all five modules was conducted by the programmer. Several software errors were discovered and corrected in each module. Unit testing was concluded when no errors were found in the last 10 manhours of testing.

Five experimental scenarios were run for the purpose of integration testing. Software errors were discovered in each test and corrected prior to the next integration test. Integration testing was limited to these five experimental scenarios due to the limited availability of student teams.

At the conclusion of the testing process, at least one software error remained uncorrected. This error consisted of a threat accelerating to a speed in excess of its maximum speed and proceeding rapidly off the screen. This error appeared to only occur when several threats and assets were clustered within a small area. This was most likely an integration error caused by the generation of several different task_evade_msgs and task_change_dir_msgs by the different heuristic models in a period of seconds.

An evasion flag for each task was added to the heuristic modules in an attempt to correct the apparent interaction problems. The task's evasion flag was initially set to false. It was set to true when the heuristic module generates an evasive or change maneuver for the task. Each heuristic module checked that the flag was false before generating a maneuver for the task. The implementation of the flag was expected to prevent the generation of more than one maneuver for a task during each update loop or each second.

The implementation of the evasion flag did reduce the occurrences of the reported software error. However, during the final integration testing trial the error occurred once. The occurrence of the error was verified using the built-in replay function of the paradigm.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The Composite Warfare Commander - Distributed Dynamic Decisionmaking (CWC-DDD) paradigm is a valuable tool in the study of the distributed decisionmaking processes. Its developers were quite successful in abstracting many of the elements of "real world" Naval engagement scenarios into the paradigm without losing too much realism. The current implementation of the paradigm does represent an acceptable balance between controllability and realism.

The current paradigm should not remain static. The addition of "intelligence" heuristics can only improve the paradigm. Each heuristic added serves to make the paradigm more realistic. The addition of each heuristic does detract very slightly from the controllability of the experiment, but the benefits are well worth the cost.

The addition of the five heuristics discussed in this paper have added to the realism of the paradigm. Real world tasks do not blindly follow a preprogrammed route to their objective. Real world tasks have certain unpredictability to their actions. The paradigm's tasks should have the same qualities. The five heuristics take some of the predictability out of the tasks, and add to the dynamics of the situation.

B. RECOMMENDATIONS

The paradigm should continue to be used as a tool for studies of distributed decisionmaking. The balance between realism and controllability guarantees meaningful results from any properly designed experiments; meaningful results that can have application to real world situations.

The composition of the subject teams for experiments with the paradigm should be examined. To date, only one experiment utilizing military subjects has been conducted. All other experiments using the paradigm have used college students as subjects. The paradigm is being used primarily for the study of distributed decisionmaking in a military environment. Few college students are familiar with living and acting in a military environment. Additional experiments should be conducted with military subjects.

The paradigm should continue to be improved through the addition of more heuristics. Each heuristic represents an addition to the realism of the model. As the realism is increased, the applicability of experimental results to the real world can only increase.

C. AREAS FOR FURTHER STUDY

1. Additional Testing of Heuristics Implemented

The five heuristics implemented in the paradigm should be subjected to a more rigorous testing program. This testing program should include the development of a test harness that will allow more effective integration testing. Given the number of software modules in the paradigm and the complexity of its

event driven environment, integration testing through exercising the paradigm produces results that are difficult to analyze and even more difficult to discover where the fault lies. A test harness would aid in isolating the errors and providing meaningful results for analysis.

2. Development of Additional Heuristics

Additional heuristics should be developed and implemented. There are numerous war game simulation programs in use at various military environments. A survey of the heuristics used in these programs would likely produce several more heuristics that could be implemented in the CWC-DDD paradigm. Each heuristic should be independently reviewed by several military commanders for validity prior to implementation.

3. Conduct Additional Experiments With Military Subjects

Additional experiments should be conducted with the CWC-DDD paradigm, both the "unintelligent" and the "intelligent" version, using military subjects. The results of experiments utilizing military subjects could be compared with the results of identical experiments conducted with civilian subjects. The comparison could provide some interesting information about the differences, or lack of differences, in distributed decisionmaking in the military and civilian populations.

4. Conversion to X Windows

Sun Windows or Sunview is Sun Microsystems' proprietary window system for their workstations. X Windows is a windowing system designed to run on most Unix based workstations. The paradigm currently runs only under

Sunview, therefore, its portability is limited to Sun or Sun compatible workstations. Converting the paradigm to X Windows would allow it to be ported to almost any Unix based workstation and would increased the availablilty of the paradigm for experimentation.

REFERENCES

Darnell, Peter A. and Philip E. Margolis. 1991. *C: A Software Engineering Approach*. New York: Springer-Verlag New York, Inc.

Kerighan, Brian W. and Dennis M. Ritchie. 1988. *The C Programming Language - Second Edition*. Englewood Cliffs, New Jersey: Prentice Hall.

Heslop, Brent D. and David Angell. 1990. *Mastering SunOS*. San Francisco: SYBEX, Inc.

Kleinman, David L., Daniel Serfaty, and Peter Luh. 1984. A Research Paradigm For Multi-Human Decision Making. In *Proceedings of the 1984 American Control Conference in San Diego, California, June 6-8, 1984*, by the American Automatic Control Council. Piscataway, New Jersey: IEEE Service Center.

Kleinman, David L. and Anlan Song. 1990. A Research Paradigm For Studying Team Decisionmaking and Coordination. In *Proceedings of the 1990 Symposium on Command and Control Research in Monterey, California, June 12-14, 1990*, by the Basic Research Group Technical Panel on C3 of the Joint Directors of Laboratories and the National Defense University. McLean, Virginia: Information Systems Division of Science Applications International Corporation.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and Williamm T. Vetterling. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. New York: Cambridge University Press.

Schildt, Herbert. 1991. *C: The Pocket Reference*. Berkeley, California: McGraw-Hill, Inc.

Schildt, Herbert. 1990. *Teach Yourself C*. Berkeley, California: McGraw-Hill, Inc.

Shi, Jian, Peter Luh, David L. Kleinman. 1990. A Normative-Descriptive Study of Information and Command Strategy in Distributed Team Resource Allocation Part 1: Experimental Design. In *Proceedings of the 1990 Symposium on Command and Control Research in Monterey, California, June 12-14, 1990*, by the Basic Research Group Technical Panel on C3 of the Joint Directors of Laboratories and the National Defense University. McLean, Virginia: Information Systems Division of Science Applications International Corporation.

Song, Anlan. 1991. CWC-DDD Software Document Part I: General Description. Internal working document. Storrs, Connecticut: Electronic Systems Engineering Department of the University of Connecticut. Photocopied.

Wu, Lining, Anlan Song. March 1990. CWC-DDD Scenario Generator Users Manual. Technical manual #90-1. Storrs, Connecticut: Electronic Systems Engineering Department of the University of Connecticut. Photocopied.

APPENDIX

ADDITIONS TO CWC-DDD CODE

Implementation of the heuristics detailed in Chapter 3 was accomplished by the addition of the following code. The code in italics is previously existing code and is included only a guide to the placement of the new code.

A. CODE ADDED TO INCLUDE/COMM_MSGS.H

```
#define TASK_PENETRATE        2018
/* Added for intelligence heuristics. */
#define TASK_EVADE            2019

    int flag;
  } swat_msg;
/* Added for intelligence heuristics. */
/*
21)       task evade
*/

          struct TaskEvade
          {
           int message_id;     /* identification */
           int task_id;        /* task identification */
           float max_v;        /* task max velocity */
           float x;            /* task current x coord */
           float y;            /* task current y coord */
           float vx;           /* task current x velocity */
           float vy;           /* task current y velocity */
           float new_x;        /* maneuver endpoint x coord */
           float new_y;        /* maneuver endpoint y coord */
           double man_time;    /* time to do maneuver */
          } task_evade_msg;

extern int xdr_swat_msg();
/* Added for intelligence heuristics. */
extern int xdr_task_evade();
```

B. CODE ADDED TO INCLUDE/DATASTRUC.H

```
asset_state_type              *pursued_by[NPLATFORM];
/* Added for intelligence heuristics. */
asset_state_type              *evading[NPLATFORM];
float                         asset_dist[NPLATFORM];
int                           evade_ctr;
int                           evade_sector;
int                           evasion;
```

C. CODE ADDED TO SRC/COMM/XDR_STRUCT.C

MODULE HISTORY   : *ORIGINAL*
                           Anlan Song   *08/06/89*
                           MODIFIED
                           Brian Wright   02/05/92

```
     // !xdr_int(xdrs, &(ptr->flag))
  ){
    fatal ("receive: cannot read data \n");
     return(FALSE);
  }
  else return TRUE;
}
/* Added for intelligence heuristics. */
/*
 21) task evade
*/
xdr_task_evade(xdrs,ptr)
  XDR *xdrs;
  struct TaskEvade *ptr;
{
  if (!xdr_int(xdrs,&(ptr->task_id))
      || !xdr_float(xdrs,&(ptr->max_v))
      || !xdr_float(xdrs,&(ptr->x))
      || !xdr_float(xdrs,&(ptr->y))
      || !xdr_float(xdrs,&(ptr->vx))
      || !xdr_float(xdrs,&(ptr->vy))
      || !xdr_float(xdrs,&(ptr->new_x))
      || !xdr_float(xdrs,&(ptr->new_y))
      || !xdr_double(xdrs,&(ptr->man_time)))
```

```
    {
      fatal("receive: cannot read TaskEvade data \n");
      return ( FALSE );
    }
      else  return TRUE;
  }
```

D. CODE ADDED TO SRC/GLOBAL/MAKEFILE

LIBS = $(LLIBS) -lsuntool -lsunwindow -lpixrect   -lm

E. CODE ADDED TO SRC/GLOBAL/GET_MSG_FROM_LOGF.C:

rec_position	*- process position message*
rec_evade	- process evade message
rec_task_chg	- process task change message

MODULE HISTORY	*: ORIGINAL*	
	Nick	*07/12/89*
	Rewrite	
	Anlan Song	*08/03/89*
	MODIFIED	
	Brian Wright	02/16/92

```
      /* if (rec_task_penetrate(dm) < 0)
            fprintf(stderr, "error in task_penetrate! \n");
      */
      break;
/* Next two cases added for intelligence heuristics. */
case TASK_EVADE:
   /* Skip task evade message, because locals will send this command
    * to global when a task needs to evade. The evade message
    * in log file is used only for data analysis, not for re-play.
    */
   /* if (rec_evade(dm) < 0)
            fprintf(stderr,"error in task_evade! \n");
   */
    break;
```

```
case TASK_CHANGE:
  /* Skip task change message, because locals will send this command
   * to global when a task changes course. The change message
   * in log file is used only for data analysis, not for re-play.
   */
  /* if (rec_task_chg(dm) < 0)
          fprintf(stderr,"error in task_change! \n");
  */
  break;
```

F. CODE ADDED TO SRC/GLOBAL/PROCESS_GLOBAL_IO.C

rec_assign_task	*- process assign_task message*
rec_evade	- process evade message
rec_task_chg	- process task change message

MODULE HISTORY	*: ORIGINAL*	
	Nick	*07/12/89*
	Rewrite	
	Anlan Song	*08/03/89*
	MODIFIED	
	Brian Wright	02/16/92

```
        fprintf(stderr, "error in rec_swat! \n");
      break;
   /* Next two case added for intelligence heuristics. */
   case TASK_EVADE:
     dm = (DM_type)find_object(read_fds, fd);
     data = (char *)&task_evade_msg;
     receive_xdr_data(readObj, TYPE_STRUCT, data,
                           (char *)xdr_task_evade);
     if (rec_evade(dm) < 0) fprintf(stderr,"error in rec_evade! \n");
     break;
   case TASK_CHANGE:
     dm = (DM_type)find_object(read_fds, fd);
     data = (char *)&task_change_dir_msg;
     receive_xdr_data(readObj, TYPE_STRUCT, data,
                           (char *)xdr_task_change);
     if (rec_task_chg(dm) < 0)
         fprintf(stderr,"error in rec_task_chg! \n");
     break;
```

60

G. CODE ADDED TO SRC/GLOBAL/READ_LOGF.C

MODULE HISTORY   : *ORIGINAL*
                                         Anlan Song   *08/06/89*
                                         MODIFIED
                                         Brian Wright   02/05/92

```
        &(task_penetrate.flag));
      break;
    /* Next two cases added for intelligence heuristics. */
    case TASK_EVADE:
      fscanf(logfp," %d %f %f %f %f %f %f %f %f\n",
        &(task_evade_msg.task_id),
        &(task_evade_msg.max_v),
        &(task_evade_msg.x),
        &(task_evade_msg.y),
        &(task_evade_msg.vx),
        &(task_evade_msg.vy),
        &(task_evade_msg.new_x),
        &(task_evade_msg.new_y),
        &(task_evade_msg.man_time));
      break;
    case TASK_CHANGE:
      fscanf(logfp," %d %f %f %f %f\n",
        &(task_change_dir_msg.task_number),
        &(task_change_dir_msg.x),
        &(task_change_dir_msg.y),
        &(task_change_dir_msg.vx),
        &(task_change_dir_msg.vy));
      break;
```

H. CODE ADDED TO SRC/GLOBAL/REC_COMMAND.C

```
#define DISAPPEAR_DELAY  15
#include <stdio.h>
#include <math.h>
```

61

```
        set_task_event(current_time+DISAPPEAR_DELAY, ptr1);
        return 0;
}
/******************************************************/
/*                          function "rec_evade"
```

FUNCTION	: This function processes evade message.	
PARAMETERS	: dm - dm identification	
RETURN	:	
LOCAL VARIABLES	: dummy_list	- temporary list to hold modified events
	prtsk, nevent	
	ptrevt, dummy	- pointers to events
	eptr1, eptr2	- pointers to TaskChange info
	eve_buf1	
	eve_buf2	- temporary pointers to certain TaskChange info
	old_event_time	
	event_time	- time of an event
	leg_time	
	aleg_time	- time to complete a maneuver
	delta_t.	- difference between original event time and new event time
	x_dist	
	y_dist	- x and y components of distance
	leg_dist	- distance of maneuver
	v	- speed required to accomplish maneuver
SUBROUTINES CALLED	: send_msg_to_buf	- send message to the send out buffer
	write_logf	- write log file
	add_ct	- set event
	remove_ct	- remove event
	del_asset_event	- delete future event related to a certain asset

```
CALLED BY              : process_local_input
MODULE HISTORY         : ORIGINAL
                         Brian Wright            01/30/92   */
/********************************************************/
rec_evade(dm)
DM_type dm;
{
 OBJ dummy_list;
 event_struct *prtsk, *ptrevt, *nevent, *dummy;
 struct TaskChange *eptr1, *eptr2;
 struct EveBuf {
  int message_id;
  int obj_id;
  } *eve_buf1, *eve_buf2;
 double event_time, leg_time, aleg_time, delta_t, old_event_time;
 double x_dist, y_dist, temp, leg_dist;
 float v;

 if (mode == 0) write_logf(dm, TASK_EVADE);
 send_msg_to_buf(TASK_EVADE);

 /* Create dummy list. */
 dummy_list = create_sequence( );

 /* Find first TASK_CHANGE event in future event stack, if any, involving
    this task. */
 for(prtsk = (event_struct *)first_ct(task_event); prtsk;
     prtsk = (event_struct *)next_ct(task_event))
 {
  eve_buf1 = (struct EveBuf *)(prtsk->eventptr);
  if (eve_buf1->obj_id == task_evade_msg.task_id &&
     eve_buf1->message_id == TASK_CHANGE)
  {
   eptr1 = (struct TaskChange *)(prtsk->eventptr);
   old_event_time = prtsk->time;
   break;
  }
 }

 /* Calculate event time for new TASK_CHANGE event. */
 event_time = current_time + task_evade_msg.man_time;
```

63

```c
/* Calculate time interval between new TASK_CHANGE event and
    TASK_CHANGE event found in future event stack. */
leg_time = old_event_time - event_time;

/* Calculate distance between new TASK_CHANGE event point and
    point of TASK_CHANGE event found in future event stack. */
x_dist = (double)(eptr1->x - task_evade_msg.new_x);
y_dist = (double)(eptr1->y - task_evade_msg.new_y);
temp = x_dist*x_dist + y_dist*y_dist;
leg_dist = sqrt(temp);

/* Calculate velocity needed to reach TASK_CHANGE event found in
    future event stack on time. */
v = (float)leg_dist / (float)leg_time;

/* Create new TASK_CHANGE event for future event list. */
eptr2 = (struct TaskChange *)Calloc(1, sizeof(task_change_dir_msg));
eptr2->message_id = TASK_CHANGE;
eptr2->task_number = task_evade_msg.task_id;
eptr2->x = task_evade_msg.new_x;
eptr2->y = task_evade_msg.new_y;

/* Check to see if v is less than max_v for task. */
if (v <= task_evade_msg.max_v)
{
  /* Calculate components of v for TASK_CHANGE event. */
  eptr2->vx = v * (float)(x_dist / leg_dist);
  eptr2->vy = v * (float)(y_dist / leg_dist);
}
else
{
  /* Calculate components of max_v for TASK_CHANGE event. */
  eptr2->vx = (task_evade_msg.max_v) * (float)(x_dist / leg_dist);
  eptr2->vy = (task_evade_msg.max_v) * (float)(y_dist / leg_dist);

  /* Calculate time required to cover leg at max_v for task. */
  aleg_time = leg_dist / (double)task_evade_msg.max_v;

  /* Calculate extra time needed to reach position for next event
      involving this task. */
  delta_t = aleg_time - (old_event_time - event_time);
}
```

```c
/* Create new event and put in dummy list. */
nevent = (event_struct *)Calloc(1, sizeof(event_struct));
nevent->time = event_time;
nevent->eventptr = (char *)eptr2;
add_ct(dummy_list, nevent);

/* Find all future events for this task. */
for(ptrevt = (event_struct *)first_ct(task_event); ptrevt;
      ptrevt = (event_struct *)next_ct(task_event))
{
  eve_buf2 = (struct EveBuf *)(ptrevt->eventptr);
  if (eve_buf2->obj_id == task_evade_msg.task_id)
  {
    nevent = (event_struct *)Calloc(1, sizeof(event_struct));

    /* Change event time if needed. */
    if (v > task_evade_msg.max_v) nevent->time = ptrevt->time + delta_t;
    if (v <= task_evade_msg.max_v) nevent->time = ptrevt->time;
    nevent->eventptr = ptrevt->eventptr;

    /* Insert new event in dummy_list. */
    add_ct(dummy_list, nevent);

    /* Remove old task events from task future event list. */
    remove_ct(task_event, ptrevt);
  }
}

/* Put all events in dummy_list in task future event list. */
for(dummy = (event_struct *)first_ct(dummy_list); dummy;
      dummy = (event_struct *)next_ct(dummy_list))
{
  if (dummy->time > current_time)
    set_task_event(dummy->time, dummy->event_ptr);
  remove_ct(dummy_list, dummy);
}
return;
}
```

65

```
/**********************************************************/
/*                          function "rec_task_chg"

FUNCTION            : This function processes task change message.
PARAMETERS          : dm - dm identification
RETURN              :
LOCAL VARIABLES     : ptrevt          - pointer to the event
                      eve_buf         - pointer to event info
SUBROUTINES CALLED  : send_msg_to_buf- send message to
                                       the send out buffer.
                      write_logf      - write log file
                      remove_ct       - delete event related to
                                        a certain asset
CALLED BY           : process_local_input
MODULE HISTORY      : ORIGINAL
                      Brian Wright          02/05/92    */
/**********************************************************/
rec_task_chg(dm)
DM_type dm;
{
  event_struct *ptrevt;
  struct EveBuf {
   int message_id;
   int obj_id;
  } *eve_buf;
  if (mode == 0) write_logf(dm, TASK_CHANGE);
  send_msg_to_buf(TASK_CHANGE);
  for(ptrevt = (event_struct *)first_ct(task_event); ptrevt;
       ptrevt = (event_struct *)next_ct(task_event))
  {
    eve_buf = (struct EveBuf *)(ptrevt->eventptr);
    if (eve_buf->obj_id == task_change_dir_msg.task_number &&
        eve_buf->message_id == TASK_CHANGE)
    {
      remove_ct(task_event, ptrevt);
      free((char *)ptrevt->eventptr);
      free((char *)ptrevt);
    }
  }
  return;
}
```

66

I. CODE ADDED TO SRC/GLOBAL/SEND_MSG_TO_BUF.C

MODULE HISTORY   : *ORIGINAL*
               Anlan Song   *08/06/89*
               MODIFIED
               Brian Wright   01/15/92

```
      ptrbuf = (struct Buff *)msgptr; ptrbuf->message_id = ASSIGN_TASK;
      add_ct(output_msg, msgptr);
      break;
/* Added for intelligence heuristics. */
case TASK_EVADE:
  msgptr = (char *)Calloc(1, sizeof(task_evade_msg));
  bcopy( (char *)&task_evade_msg, msgptr, sizeof(task_evade_msg));
  ptrbuf = (struct Buff *)msgptr; ptrbuf->message_id = TASK_EVADE;
  add_ct(output_msg, msgptr);
  break;
```

J. CODE ADDED TO SRC/GLOBAL/TIME_EVENTS.C FUNCTION "SEND_MSG_TO_LOCAL"

MODULE HISTORY   : *ORIGINAL*
               Anlan Song   *08/06/89*
               MODIFIED
               Brian Wright   01/15/92

```
      send_xdr_msg_and_data(writeObj, ASSIGN_TASK, TYPE_STRUCT,
                                data, (char *)xdr_assign_task);
/* Added for intelligence heuristics. */
case TASK_EVADE:
  for(dm = (int)DM0; dm < (int)DM_LAST; dm++){
    writeObj = get_object_at(fd_to_xdrs,
                    get_object_at_index(write_fds, dm));
    if (writeObj)      /* if this player is in... */
      send_xdr_msg_and_data(writeObj, TASK_EVADE, TYPE_STRUCT,
                                data, (char *)xdr_task_evade);
  }
  break;
```

K. CODE ADDED TO SRC/GLOBAL/WRITE_LOGF.C

MODULE HISTORY   : *ORIGINAL*
                                     Anlan Song   *08/06/89*
                                     MODIFIED
                                     Brian Wright   01/15/92

```
        task_penetrate.flag);
      break;
    /* Next two cases added for intelligence heuristics. */
    case TASK_EVADE:
      fprintf(logfp,"%d %d  %lf\n %d  %f %f %f %f %f %f %f %f\n",
        dm, TASK_EVADE, current_time,
        task_evade_msg.task_id,
        task_evade_msg.max_v,
        task_evade_msg.x,
        task_evade_msg.y,
        task_evade_msg.vx,
        task_evade_msg.vy,
        task_evade_msg.new_x,
        task_evade_msg.new_y,
        task_evade_msg.man_time);
      break;
    case TASK_CHANGE:
      fprintf(logfp,"%d %d  %lf\n %d %f %f %f %f\n",
        dm, TASK_CHANGE, current_time,
        task_change_dir_msg.task_number,
        task_change_dir_msg.x,
        task_change_dir_msg.y,
        task_change_dir_msg.vx,
        task_change_dir_msg.vy);
      break;
```

L. CODE ADDED TO SRC/LOCAL/PROCESS_LOCAL_IO.C

```
        receive_assign_task( &assign_task );
        break;
   /* Added for intelligence heuristics. */
   case TASK_EVADE:
     data = (char *)&task_evade_msg;
     receive_xdr_data(readObj, TYPE_STRUCT, data,
                         (char *)xdr_task_evade);
     task_evade( &task_evade_msg );
     break;
```

M. CODE ADDED TO SRC/LOCAL_LIB/INITIALIZE.C

```
        task_states[i].fusion_confidence = 0.0;
        /* Added for intelligence heuristics. */
        task_states[i].evade_ctr = -1;
        task_states[i].evade_sector = -1;
        task_states[i].evasion = FALSE;
```

N. CODE ADDED TO SRC/LOCAL_LIB/RECEIVE.C

```
   printf("task destroyed id=%d\n",task->id);
     /* display_fb_info(ptr->score); */
   }
   /***********************************************************/
   /*                        function "task_evade"
```

FUNCTION	: evade an asset
INPUT VARIABLES	:
OUTPUT VARIABLES	: Update the position and velocity of the task based upon the task_evade_msg.
LOCAL VARIABLES	: x, y - x and y coordinates of task position
	task_id - id of task
	task - pointer to task
SUBROUTINES CALLED	:
CALLED BY	: receive.c
MODULE HISTORY	: ORIGINAL
	Brian Wright 01/15/92 */

```
   /***********************************************************/
```

69

```
task_evade()
{
  extern OBJ tasks;
  task_state_type *task;
  int task_id;
  float x, y;

  /* Find the task in the task sequence. */
  task_id = task_evade_msg.task_id;
  for(task = (task_state_type *)first_ct(tasks); task;
        task = (task_state_type *)next_ct(tasks))
    if (task->id == task_id) break;
  if (!task){
    fprintf(stderr, "task_evade_msg: task %d not exist!\n", task_id);
    return(1);
  }

  /* Update the coordinates and velocity of the task. */
  task->x = task_evade_msg.x;
  task->y = task_evade_msg.y;
  task->vx = task_evade_msg.vx;
  task->vy = task_evade_msg.vy;
  task_id = task->id;
  x = task->x;
  y = task->y;
  printf("task_evade: task %d\t x %f\t y %f\n", task_id, x, y);
  return;
}
```

O. CODE ADDED TO SRC/LOCAL_LIB/SEND_MSG_TO_GLOBAL.C

```
/* send message to global */
/* MODIFIED  Brian Wright  2/16/92 */
```

```
send_xdr_msg_and_data(writeObj, SWAT_MSG, TYPE_STRUCT,
                          data, (char *)xdr_swat_msg);
  break;
/* The next two cases were added as part of the intelligence heuristics
    for the tasks. */
case TASK_EVADE:
  data = (char *)&task_evade_msg;
  send_xdr_msg_and_data(writeObj, TASK_EVADE, TYPE_STRUCT,
                          data, (char *)xdr_task_evade);
  break;
case TASK_CHANGE:
  data = (char *)&task_change_dir_msg;
  send_xdr_msg_and_data(writeObj, TASK_CHANGE, TYPE_STRUCT,
                          data, (char *)xdr_task_change);
  break;
```

P. CODE ADDED TO SRC/LOCAL_LIB/UPDATE.C

is set wherever there is a communication from one DM to another.

The following 5 heuristics were added to introduce a rudimentary intelligence to the actions of the tasks.

- The function checks to see if the number of tasks within a sector is exceeded by the number of assets. The counts are maintained by an enemy command and control task located on the primary attack axis just outside DM0's central platform's detection range. The C2 task has a detection range of 0.5. If the number of tasks is exceeded by the number of assets, a course change to the neighboring sector with the smaller number of assets is generated for each task.

- The function checks to see if there are sufficient resources enroute a task to properly destroy it. If there are, an evasive maneuver is generated for the task.

- The function checks to see if the task is within the weapons range of any asset. If it is, an evasive maneuver is generated for the task.

- The function checks to see if a primary attack task is within the inner radius. If it is, it accelerates to max_v and turns directly for the center.

71

- The function checks to see if a diversionary attack task is inside all the assets. If it is, it turns towards the center at max_v and becomes an attacker rather than veering off.

equal to pursue_parameter.
att_penet_parameter
- a design parameter that is constant. After every att_penet_parameter times certain functions are called.

att_penet_loop_number
- is incremented every time update is called until equal to att_penet_parameter.

penetrated inside the (penetration) zone.
attempt_penetration
- it is called to change the course of a primary attack task directly for the center when it reaches the inner radius

change_to_attacker
- it is called to change the course of a diversionary attacker for the center rather than veering off if it gets inside all assets

sector_check
- it is called to generate evasive maneuvers for tasks when they are outnumbered in their sector

resources_on_way
- it is called to generate an evasive maneuver for a task when sufficient resources are enroute to properly destroy it

MODULE HISTORY   : *ORIGINAL*
                         Paiman Nodoushani   *08/11/89*
                         MODIFIED
                         Brian Wright   02/16/92

```c
int pursue_flg;
static int att_penet_loop_num=1;
static int att_penet_parameter=4;
int att_penet_flg;

 pursue_loop_number++;
}
/* set att_penet_flg */
if (att_penet_loop_num == att_penet_parameter){
 att_penet_flg = TRUE;
 att_penet_loop_num = 1;
} else {
 att_penet_flg = FALSE;
 att_penet_loop_num++;
}


/* Update all tasks. */
/* Check if any tasks are outnumbered in their sector. */
if (this_dm == DM0) sector_check( );

update_responsibility(task_ptr);;
/* Execute these actions only once every five seconds, if task
    is a threat, and if an evasive maneuver is not pending. */
if (att_penet_flg && is_threat(task_ptr))
{
  if(task_ptr->state != DESTROYED &&
     task_ptr->state != DISAPPEARED &&
     this_dm == DM1)
  {
    /* Check to see if there are sufficient resources enroute to
        properly destroy this task. */
    if (!task_ptr->evasion) resources_on_way(task_ptr);

    /* If this task is part of the primary attack and has reached the
        inner radius, have it turn for the center at max_v. */
    if (!task_ptr->evasion) attempt_penetration(task_ptr);

    /* If this task is part of the diversionary attack and is inside
        all of the assets, have it turn towards the center and attack. */
    if (!task_ptr->evasion) change_to_attacker(task_ptr);
  }
}
task_ptr->evasion = FALSE;
```

```
int acquire_ctr = 0;
int type;
 int evaded_flg = 0;
int i, index1, index2, index3;

    task_ptr->acquired_by[acquire_ctr] = asset_ptr;
    acquire_ctr++;
    }
  if(is_threat(task_ptr)) /* Execute for threats only. */
  {

    /* If the task is within the weapon range of a platform and the task
        is not already evading this asset, have DM0 calculate an evasive
        maneuver for this task */
    if (distance <= asset_ptr->asset_class->weapon_range[type] &&
        this_dm == DM0)
    {
      for(index1 = 0; index1 <= task_ptr->evade_ctr; index1++)
       if (task_ptr->evading[index1] == asset_ptr) evaded_flg = 1;
      if (evaded_flg != 1)
      {
        /* Update list of assets being evaded by this task. */
        task_ptr->evade_ctr++;
        task_ptr->evading[task_ptr->evade_ctr] = asset_ptr;
        task_ptr->asset_dist[task_ptr->evade_ctr] = distance;

        /* Generate evasive maneuver for this task if no other evasive
            maneuvers are pending for this task. */
        if (!task_ptr->evasion)
          within_weapons_range(task_ptr, asset_ptr, distance);
      }
    }
```
74

```
        /* Update list of assets being evaded by this task. */
        if (distance > asset_ptr->asset_class->weapon_range[type] &&
            this_dm == DM0)
       {
         for(index2 = 0; index2 <= task_ptr->evade_ctr; index2++)
         {
           if (task_ptr->evading[index2] == asset_ptr)
           {
             for(index3 = index2; index3 <= task_ptr->evade_ctr-1; index3++)
             {
               task_ptr->evading[index3] = task_ptr->evading[index3+1];
               task_ptr->asset_dist[index3] = task_ptr->asset_dist[index3+1];
             }
             task_ptr->evading[task_ptr->evade_ctr] = NULL;
             task_ptr->asset_dist[task_ptr->evade_ctr] = 0;
             task_ptr->evade_ctr--;
           }
         }
       }
     }


     if (task_ptr->penetrate_flag == PENETRATING)
        task_ptr->penetrate_flag = PENETRATED);
}
/*****************************\*********************************************/
/*                           function "within_weapons_range"
```

FUNCTION	: This function calculates an evasive maneuver for a threat that has been found to be inside the weapons range of an asset but is outside the inner radius.
INPUT VARIABLES	: tptr - pointer to task being checked
	aptr - pointer to asset being evaded
	d - distance between task and asset
OUTPUT VARIABLES	: The function generates a task_evade_msg for the task that meets the conditions above, and sends the message to global for action.

```
GLOBAL VARIABLES        : tasks
                          platforms          - sequences containing the
                                               pointers to tasks and assets
LOCAL VARIABLES         : dtcx, dtcy
                          dx, dy             - x and y components of
                                               distance between two
                                               points
                          dtc                - distance between two
                                               points
                          angle
                          rangle             - angle between course
                                               line and x axis
                          test_val           - random number used to
                                               determine whether course
                                               change should be + or -
                          rdist              - distance for evasive
                                               maneuver
                          man_time           - time to do evasive
                                               maneuver
CALLED BY               : update_task_state
SUBROUTINES CALLED      : unirand            - calculates uniformly
                                               distributed random
                                               number
                          penetration_r      - calculates inner radius
MODULE HISTORY          : ORIGINAL
                          Brian Wright            1/15/92            */
/**********************************************************************/
within_weapons_range(tptr, aptr, d)
task_state_type *tptr;
asset_state_type *aptr;
float d;
{
  double unirand(), penetration_r();
  double dx, dy, angle, rangle, test_val;
  double dtcx, dtcy, tctemp, dtc;
  double rdist, temp, man_time;

  /* If within inner radius, do not do evasive maneuver. */
  dtcx = (double)(0.5 - tptr->x);
  dtcy = (double)(0.5 - tptr->y);
  tctemp = dtcx*dtcx + dtcy*dtcy;
  dtc = sqrt(tctemp);
  if (dtc <= penetration_r()) return;
```

```c
/* Calculate angle between positive horizontal axis and line
    connecting task and asset positions. */
dx = (double)(aptr->x - tptr->x);
dy = (double)(aptr->y - tptr->y);
angle = atan2(dy, dx);

/* Calculate random angle for task to change course relative to
    the asset. */
test_val = unirand();
if(test_val <= 0.5)
  rangle = (unirand() * 1.570796327) + angle;
else
  rangle = (unirand() * -1.570796327) + angle;

/* Check if random angle greater than 360 degrees or less
    than 0 degrees. */
if (rangle >= 6.283185307) rangle = rangle - 6.283185307;
if (rangle < 0.0) rangle = rangle + 6.283185307;

/* Calculate random distance for evasive maneuver. */
rdist = unirand() * (double)d;

/* Calculate time to accomplish evasive maneuver. */
man_time = rdist / (double)(tptr->task_class->max_v);

/* Send information to global for processing. */
task_evade_msg.task_id = tptr->id;
task_evade_msg.max_v = tptr->task_class->max_v;
task_evade_msg.x = tptr->x + (tptr->vx * constants.renew_interval);
task_evade_msg.y = tptr->y + (tptr->vy * constants.renew_interval);
task_evade_msg.vx = (tptr->task_class->max_v) * (float)cos(rangle);
task_evade_msg.vy = (tptr->task_class->max_v) * (float)sin(rangle);
task_evade_msg.new_x = (float)(rdist * cos(rangle)) + tptr->x;
task_evade_msg.new_y = (float)(rdist * sin(rangle)) + tptr->y;
task_evade_msg.man_time = man_time;
send_msg_to_global(TASK_EVADE);
tptr->evasion = TRUE;
return;
}
```

```
/***********************************************************/
/*                        function "attempt_penetration"

    FUNCTION            : This function determines if a task is in the
                           primary attack and if the task is within the
                           first circle drawn outside the penetration
                           zone (the inner radius).  If it is, the task
                           turns for the center at maximum velocity.
    INPUT VARIABLES     : tptr            - pointer to task being
                                              checked
    OUTPUT VARIABLES    : The function generates a
                           task_change_dir_msg for the task that meets
                           the conditions above and sends the message
                           to global for action.
    GLOBAL VARIABLES    : constants.pri_att  - primary attack sector
    LOCAL VARIABLES     : in_sector        - sector task is currently in
                           dx, dy           - x and y components of
                                              distance between two
                                              points
                           d                - distance between two
                                              points
    CALLED BY           : update
    SUBROUTINES CALLED  : xy2sector        - determines which sector
                                              task is in
                           penetration_r   - determines inner radius
    MODULE HISTORY      : ORIGINAL
                           Brian Wright          2/05/92      */
/***********************************************************/
attempt_penetration(tptr)
task_state_type *tptr;
{
  int in_sector;
  double dx, dy, temp, d;
  double penetration_r();

  /* Determine which sector the task is in. */
  in_sector = xy2sector(tptr->x, tptr->y);

  /* Determine if the task is in the primary attack axis sector. */
  if(in_sector == constants.pri_att)
  {
```

```
/* Calculate distance between task position and center. */
dx = (double)(0.5 - tptr->x);
dy = (double)(0.5 - tptr->y);
temp = dx*dx + dy*dy;
d = sqrt(temp);

/* If task is within inner radius, have task turn directly for
   penetration zone. */
if (d <= penetration_r())
{
  task_change_dir_msg.message_id = TASK_CHANGE;
  task_change_dir_msg.task_number = tptr->id;
  task_change_dir_msg.x = tptr->x + (tptr->vx *
          constants.renew_interval);
  task_change_dir_msg.y = tptr->y + (tptr->vy *
          constants.renew_interval);
  task_change_dir_msg.vx = tptr->task_class->max_v * (float)(dx / d);
  task_change_dir_msg.vy = tptr->task_class->max_v * (float)(dy / d);
  send_msg_to_global(TASK_CHANGE);
  tptr->evasion = TRUE;
  }
 }
 return;
}
```

/***/
/*   function "change_to_attacker"

FUNCTION   : This function determines if a task is in the
                         diversionary attack and if there are any assets
                         closer to the center than the task. If there
                         are none, the task becomes an attacker and
                         turns for the center at maximum velocity.
INPUT VARIABLES   : tptr   - pointer to task being
                                                checked
OUTPUT VARIABLES   : The function generates a
                         task_change_dir_msg for the task that meets
                         the conditions above and sends the message
                         to global for action.
GLOBAL VARIABLES   : constants.div_att - diversionary attack sector
                         platforms   - sequence containing the
                                              pointers to assets

79
```

| LOCAL VARIABLES | : | in_sector adx, ady | - sector task is currently in |
|---|---|---|---|
| | | tdx, tdy | - x and y components of distance between two points |
| | | ad, td | - distance between two points |
| | | asset_ptr | - pointer to asset |
| | | chg_flg | - flag used to determine whether task_change_dir _msg should be sent |
| CALLED BY | : update | | |
| SUBROUTINES CALLED | : xy2sector | | - detemines which sector task is in |
| MODULE HISTORY | : ORIGINAL Brian Wright | | 2/05/92 */ |

```
/*****************************************************/
change_to_attacker(tptr)
task_state_type *tptr;
{
  asset_state_type *asset_ptr;
  int chg_flg = TRUE;
  int in_sector;
  double tdx, tdy, ttemp, td;
  double adx, ady, atemp, ad;

  /* Determine which sector the task is in. */
  in_sector = xy2sector(tptr->x, tptr->y);

  /* Determine if the task is in the diversionary attack axis sector. */
  if(in_sector == constants.div_att)
  {
    /* Calculate distance between task position and center. */
    tdx = (double)(0.5 - tptr->x);
    tdy = (double)(0.5 - tptr->y);
    ttemp = tdx*tdx + tdy*tdy;
    td = sqrt(ttemp);

    /* Determine if there are any assets closer to the center than
       this task. */
    for(asset_ptr = (asset_state_type *)first_ct(platforms); asset_ptr;
        asset_ptr = (asset_state_type *)next_ct(platforms))
    {
```

```
    if (asset_ptr->x != 0.5 && asset_ptr->y != 0.5)
    {
      adx = (double)(0.5 - asset_ptr->x);
      ady = (double)(0.5 - asset_ptr->y);
      atemp = adx*adx + ady*ady;
      ad = sqrt(atemp);
      if (ad <= td) chg_flg = FALSE;
    }
  }

  /* If there are no assets closer to the center than this task,
       change the course of the task towards the center. */
  if (chg_flg)
  {
   task_change_dir_msg.message_id = TASK_CHANGE;
    task_change_dir_msg.task_number = tptr->id;
    task_change_dir_msg.x = tptr->x + (tptr->vx *
            constants.renew_interval);
    task_change_dir_msg.y = tptr->y + (tptr->vy *
            constants.renew_interval);
    task_change_dir_msg.vx = tptr->task_class->max_v * (float)(tdx / td);
    task_change_dir_msg.vy = tptr->task_class->max_v * (float)(tdy / td);
    send_msg_to_global(TASK_CHANGE);
    tptr->evasion = TRUE;
   }
 }
 return;
}


/****************************************************/
/*                          function "sector_check"

FUNCTION                  : This function counts the number of assets in
                            each sector visible to a task command and
                            control platform located on the primary
                            attack axis just outside of DM0's central
                            platform's detection range.  The detection
                            range of this enemy C2 platform is a circle
                            of radius 0.5.  If the number of assets
                            detected within a sector exceeds the number
                            of tasks within that sector, the tasks will
                            change course to the neighboring sector with
                            the fewer detected assets.
```

INPUT VARIABLES	:	
OUTPUT VARIABLES	:	The function generates task_evade_msg for each task that requires a course change under the conditions above. The message is sent to global for action.
GLOBAL VARIABLES	:	tasks
		platforms — sequences containing the pointers to the tasks and assets
LOCAL VARIABLES	:	numx
		numy — arrays with the x and y coordinates for the position of the enemy command and control platform for each sector
		asset_count
		task_count — arrays of counts of number of assets and tasks in each sector
		sector — sector task or asset is currently in
		dx, dy — x and y components of distance between two points
		d — distance between two points
		tc_dist — distance between task and center
		nc_dist — distance between new point and center
		tn_dist — distance between new point and task position
		theta1
		theta2 — angles between course line and x axis
		man_time — time required for evasive maneuver
		asset_ptr
		tk_ptr
		task_ptr — pointers to task or asset
		new_x
		new_y — coordinates of new destination point

```
CALLED BY              : update
SUBROUTINES CALLED     : unirand      - calculates uniformly
                                        distributed random
                                        number
                         xy2sector    - determines what sector
                                        object is in
MODULE HISTORY         : ORIGINAL
                         Brian Wright          2/15/92      */
/****************************************************************/
sector_check()
{
  static float numx[12] = { 0.750000, 0.933013, 1.005000, 0.933013,
                            0.760000, 0.500000, 0.230000, 0.056987,
                           -0.015000, 0.056987, 0.240000, 0.500000 };
  static float numy[12] = { 0.066987, 0.250000, 0.500000, 0.760000,
                            0.943013, 1.000000, 0.943013, 0.760000,
                            0.500000, 0.230000, 0.059987, 0.000000 };
  int asset_count[12], task_count[12];
  int i, sector;
  float new_x, new_y;
  double dx, dy, temp, d;
  double tc_dist, nc_dist, tn_dist;
  double theta1, theta2, man_time;
  asset_state_type *asset_ptr;
  task_state_type *task_ptr, *tk_ptr;
  double unirand();

  /* Initialize asset and task counts for each sector to 0. There are 12
       sectors, numbered from 0 to 11. */
  for(i = 0; i < 12; i++)
  {
    asset_count[i] = 0;
    task_count[i] = 0;
  }

  /* Count number of assets in each sector visible to an enemy command
       and control platform.  The detection range of this enemy platform
       is a circle of radius 0.5.  The enemy platform is located at a point
       on the primary attack axis just outside of DM0's central platform's
       detection range.  Ignore asset in center since it is not in a specific
       sector. */
  for(asset_ptr = (asset_state_type *)first_ct(platforms); asset_ptr;
       asset_ptr = (asset_state_type *)next_ct(platforms))
```

```
{
  if (asset_ptr->x != 0.5 && asset_ptr->y != 0.5)
  {
    dx = numx[constants.pri_att-1] - asset_ptr->x;
    dy = numy[constants.pri_att-1] - asset_ptr->y;
    temp = dx*dx + dy*dy;
    d = sqrt(temp);
    if (d <= 0.5)
    {
      sector = xy2sector(asset_ptr->x, asset_ptr->y);
      asset_count[sector-1]++;
    }
  }
}

/* Count number of tasks in each sector visible to an enemy command
   and control platform.  The detection range of this enemy platform
   is a circle of radius 0.5.  The enemy platform is located at a point
   on the primary attack axis just outside of DM0's central platform's
   detection range. */
for(task_ptr = (task_state_type *)first_ct(tasks); task_ptr;
    task_ptr = (task_state_type *)next_ct(tasks))
{
  dx = numx[constants.pri_att-1] - task_ptr->x;
  dy = numy[constants.pri_att-1] - task_ptr->y;
  temp = dx*dx + dy*dy;
  d = sqrt(temp);
  if (d <= 0.5)
  {
    sector = xy2sector(task_ptr->x, task_ptr->y);
    task_count[sector-1]++;
  }
}

/* For each task in a sector where the number of assets exceeds the
   number of tasks, generate a course change to the adjoining sector
   with the fewest assets. */
for(tk_ptr = (task_state_type *)first_ct(tasks); tk_ptr;
    tk_ptr = (task_state_type *)next_ct(tasks))
{
  /* Generate evasive maneuvers only for tasks that are threats and do not
     already have an evasive maneuver pending for them. */
  if(is_threat(tk_ptr) && !tk_ptr->evasion)
```

```
{
  sector = xy2sector(tk_ptr->x, tk_ptr->y);
  if (tk_ptr->evade_sector == sector) return;
  if (asset_count[sector-1] > task_count[sector-1])
  {
    /* Record current sector to prevent multiple evasions
        in same sector. */
    tk_ptr->evade_sector = sector;

    /* Find distance between task and center of display. */
    dx = (double)(0.5 - tk_ptr->x);
    dy = (double)(0.5 - tk_ptr->y);
    temp = dx*dx + dy*dy;
    tc_dist = sqrt(temp);

    /* Determine the coordinates of a point in the appropriate
        adjoining sector that is at a distance (from the center of the
        display) less than the current distance from the center. */
    nc_dist = tc_dist * unirand();
    if (asset_count[sector-2] < asset_count[sector])
      theta1 = 0.523598775 * (double)(sector - 2) - 1.047197551;
    else
      theta1 = 0.523598775 * (double)(sector) - 1.047197551;
    if (theta1 > 3.141592654) theta1 -= 6.283185307;
    new_x = (float)(nc_dist * cos(theta1)) + 0.5;
    new_y = (float)(nc_dist * sin(theta1)) + 0.5;

    /* Calculate distance between current position of task and new
        point.  Also calculate angle with positive x axis for this course
        line and time required to accomplish this maneuver at max_v for
        this task. */
    dx = (double)(new_x - tk_ptr->x);
    dy = (double)(new_y - tk_ptr->y);
    temp = dx*dx + dy*dy;
    tn_dist = sqrt(temp);
    theta2 = atan2(dy, dx);
    man_time = tn_dist / (double)(tk_ptr->task_class->max_v);
```

```
/* Send information to global. */
task_evade_msg.task_id = tk_ptr->id;
task_evade_msg.max_v = tk_ptr->task_class->max_v;
task_evade_msg.x = tk_ptr->x + (tk_ptr->vx *
        constants.renew_interval);
task_evade_msg.y = tk_ptr->y + (tk_ptr->vy *
        constants.renew_interval);
task_evade_msg.vx = (tk_ptr->task_class->max_v) * (float)cos(theta2);
task_evade_msg.vy = (tk_ptr->task_class->max_v) * (float)sin(theta2);
task_evade_msg.new_x = new_x;
task_evade_msg.new_y = new_y;
task_evade_msg.man_time = man_time;
send_msg_to_global(TASK_EVADE);
tk_ptr->evasion = TRUE;
    }
  }
 }
 return;
}


/*******************************************************/
/*                        function "resources_on_way"
```

FUNCTION	: This function determines if the resources enroute to a task are >= to the resources needed to properly attack the task. If they are, an evasive maneuver is generated for the task. The evasive maneuver will be at an angle of + or - 90 degrees from the current course heading, for a distance between 0 and 0.1, and at maximum velocity for this task.
INPUT VARIABLES	: tptr - pointer to task of concern
OUTPUT VARIABLES	: The function generates a task_evade_msg that is sent to global for action.
GLOBAL VARIABLES	: platforms - sequence containing the pointers to the assets
LOCAL VARIABLES	: vel_slope - slope of asset velocity vector
	a_to_t_slope - slope of directed line segment from the asset to the task

	xrun, yrise	- x and y components of a_to_t_slope
	xrun_sign	
	yrise_sign	
	vx_sign	
	vy_sign	- signs of appropriate x and y components of slopes (needed because program involves directed line segments not lines)
	lower_limit	
	upper_limit	- limits of range that a_to_t_slope must fall in for the resources of asset to be considered enroute
	angle	- angle task course line makes with x axis
	rangle	- random angle for evasive course change
	rdist	- distance for evasive maneuver
	man_time	- time evasive maneuver will take at max_v
	test_val	- random number used to determine whether evasive maneuver angle should be + or -
	total_weapon_strength	- array to hold values of resources enroute a task
	res_required	- amount of resources required to properly destroy a task
	asset_ptr	
	tptr	- pointer to task or asset
CALLED BY	: update	
SUBROUTINES CALLED	: unirand	- calculates uniformly distributed random number
MODULE HISTORY	: ORIGINAL	
	Brian Wright	2/15/92 */

```
/*************************************************************/
```

87

```c
resources_on_way(tptr)
task_state_type *tptr;
{
  asset_state_type *asset_ptr;
  double vel_slope, a_to_t_slope, yrise, xrun;
  double yrise_sign, xrun_sign, vy_sign, vx_sign;
  double lower_limit, upper_limit, angle;
  double rangle, rdist, man_time, test_val;
  double unirand();
  float total_weapon_strength[NRES];
  int i, res_required;

  /* Initialize array to zero. */
  for(i = 0; i < dimensions.nres; i++)
    total_weapon_strength[i] = 0;

  for(asset_ptr = (asset_state_type *)first_ct(platforms); asset_ptr;
        asset_ptr = (asset_state_type *)next_ct(platforms))
  {
    /* Calculate slope of velocity vector of the asset and the signs
       of the components of the slope. */
    vel_slope = (double)(asset_ptr->vy / asset_ptr->vx);
    vy_sign = (double)(asset_ptr->vy) / fabs((double)(asset_ptr->vy));
    vx_sign = (double)(asset_ptr->vx) / fabs((double)(asset_ptr->vx));

    /* Calculate slope of a directed line segment connecting the asset
       and task positions.  Calculate the signs of the components of
       the slope. */
    yrise = (double)(tptr->y - asset_ptr->y);
    xrun = (double)(tptr->x - asset_ptr->x);
    a_to_t_slope = yrise / xrun;
    yrise_sign = yrise / fabs(yrise);
    xrun_sign = xrun / fabs(xrun);

    /* Determine if the slopes of the velocity vector and the directed
       line segment are within 5% of each other.  Determine if the signs
       of the components are equal.  If both are, add the weapon strength
       of the asset to the total weapon strength that is enroute
       to the task. */
    lower_limit = vel_slope - (0.05 * vel_slope);
    upper_limit = vel_slope + (0.05 * vel_slope);
    if (a_to_t_slope >= lower_limit && a_to_t_slope <= upper_limit)
    {
```

88

```
      if (yrise_sign == vy_sign && xrun_sign == vx_sign)
      {
        for(i = 0; i < dimensions.nres; i++)
        {
          total_weapon_strength[i] += asset_ptr->strength_coef[i] *
            asset_ptr->asset_class->weapon_strength[i] *
            asset_ptr->number_of_pltfm;
        }
      }
    }
}

/* Determine if the total resources enroute to the task are equal to
    or greater than the resources required to properly destroy it.
    If they are, initiate an evasive maneuver. */
if (tptr->task_class->class_id == 1) res_required = 1;
else if (tptr->task_class->class_id == 2) res_required = 2;
else if (tptr->task_class->class_id == 3) res_required = 3;
if (total_weapon_strength[0] >= res_required)
{

  /* Calculate random angle for task to change course. */
  angle = atan2( (double)(tptr->vy), (double)(tptr->vx));
  test_val = unirand();
  if (test_val <= 0.5)
    rangle = (unirand() * 1.570796327) + angle;
  else
    rangle = (unirand() * -1.570796327) + angle;

  /* Calculate random distance for evasive maneuver. */
  rdist = unirand() * 0.05;

  /* Calculate time to accomplish random maneuver. */
  man_time = rdist / (double)(tptr->task_class->max_v);

  /* Send information to global for processing. */
  task_evade_msg.task_id = tptr->id;
  task_evade_msg.max_v = tptr->task_class->max_v;
  task_evade_msg.x = tptr->x + (tptr->vx * constants.renew_interval);
  task_evade_msg.y = tptr->y + (tptr->vy * constants.renew_interval);
  task_evade_msg.vx = (tptr->task_class->max_v) * (float)cos(rangle);
  task_evade_msg.vy = (tptr->task_class->max_v) * (float)sin(rangle);
  task_evade_msg.new_x = (float)(rdist * cos(rangle)) + tptr->x;
```

```c
     task_evade_msg.new_y = (float)(rdist * sin(rangle)) + tptr->y;
     task_evade_msg.man_time = man_time;
     send_msg_to_global(TASK_EVADE);
     tptr->evasion = TRUE;
   }
   return;
}
```

```
/*****************************************************/
/*                            function "unirand"
```

FUNCTION	: This function is called to generate a uniformly distributed random number between 0 and 1.
INPUT VARIABLES	:
OUTPUT VARIABLES	: The function is set equal to the random number.
LOCAL VARIABLES	: seed — the input value for library function srandom()
	seedbuf — a variable used to force calling of library function srandom( ) initially
	a — the random number
CALLED BY	: within_weapons_range, sector_check, resources_on_way
SUBROUTINES CALLED	: srandom, random
MODULE HISTORY	: ORIGINAL
	Brian Wright 1/10/92 */

```
/*****************************************************/
double unirand()
{
  double a;
  static int seed = 1;
  static int seedbuf = 0;
```

```
  if (seedbuf == 0){
    srandom(seed);
    seedbuf ++;
  }
  a = random() / 2147483647.0;
  seed ++;
  return(a);
}
```

```
/****************************************************/
/*                    function "penetration_r"
```

FUNCTION	: This function is called to determine the radius of the first circle drawn on the screen outside the penetration zone (the inner radius).	
INPUT VARIABLES	:	
OUTPUT VARIABLES	: The function is set equal to the inner radius.	
LOCAL VARIABLES	: sectors	- local copy of the environment variable NUM_OF_CIRCLES
	check_env	- a flag variable to stop program from checking environment variable NUM_OF_CIRCLES more than once
	horizontal_r	- half the width of a rectangular penetration zone
	vertical_r	- half the height of a rectangular penetration zone
	penet_r	- larger of horizontal_r and vertical_r
	r	- radius of penetration zone
CALLED BY	: within_weapons_range, attempt_penetration	
SUBROUTINES CALLED	:	
MODULE HISTORY	: ORIGINAL	
	Brian Wright 2/16/92 */	

```
/****************************************************/
```

91

```c
double penetration_r()
{
  double r;
  static int sectors = 4;
  static int check_env = 1;
  float horizontal_r, vertical_r, penet_r;

  /* Determine type of penetration zone and appropriate radius. */
  if (penetrate_zone.flag == RECTANGLE)
  {
    /* Rectangular penetration zone case */
    horizontal_r = penetrate_zone.w / 2.0;
    vertical_r = penetrate_zone.h / 2.0;
    if (horizontal_r >= vertical_r) penet_r = horizontal_r;
    if (vertical_r > horizontal_r) penet_r = vertical_r;
  } else {
    /* Circular penetration zone case */
    penet_r = penetrate_zone.r;
  }

  /* Calculate radius of first circle drawn outside penetration
     zone (inner radius). */
  if(check_env) /* Only check environment once! */
  {
    if(getenv("NUM_OF_CIRCLES"))
      sectors = atoi(getenv("NUM_OF_CIRCLES"));
    check_env = 0;
  }
  r = (double)(0.5 - penet_r) / (double)sectors + (double)penet_r;
  return(r);
}
```

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, California 93943-5000

3. Commandant (G-TPR-2) 2
   U. S. Coast Guard
   2100 2nd Street SW
   Washington, D. C. 20593-0001

4. Kishore Sengupta, Code AS/Se 1
   Naval Postgraduate School
   Monterey, California 93943-5000

5. Carl R. Jones, Code AS/Js 1
   Naval Postgraduate School
   Monterey, California 93943-5000

6. Tung X. Bui, Code AS/Bd 1
   Naval Postgraduate School
   Monterey, California 93943-5000

7. LT Brian Wright, USCG 1
   Superintendent (db)
   U. S. Coast Guard Academy
   15 Mohegan Avenue
   New London, Connecticut 06320-4195